PeopleSoft®

PeopleTools 8.42 Development Tools PeopleCode Reference

November 2002

PeopleTools 8.42
Development Tools
PeopleCode Reference
SKU TOOLS842PCR-B 1102

PeopleBooks Contributors: Teams from PeopleSoft Product Documentation and Development.
Copyright 1988-2002 PeopleSoft, Inc. All rights reserved.

Printed in the United States.

All material contained in this documentation is proprietary and confidential to PeopleSoft, Inc. ("PeopleSoft"), protected by copyright laws and subject to the nondisclosure provisions of the applicable PeopleSoft agreement. No part of this documentation may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, including, but not limited to, electronic, graphic, mechanical, photocopying, recording, or otherwise without the prior written permission of PeopleSoft.

This documentation is subject to change without notice, and PeopleSoft does not warrant that the material contained in this documentation is free of errors. Any errors found in this document should be reported to PeopleSoft in writing.

The copyrighted software that accompanies this document is licensed for use only in strict accordance with the applicable license agreement which should be read carefully as it governs the terms of use of the software and this document, including the disclosure thereof.

PeopleSoft, PeopleTools, PS/nVision, PeopleCode, PeopleBooks, PeopleTalk, and Vantive are registered trademarks, and Pure Internet Architecture, Intelligent Context Manager, and The Real-Time Enterprise are trademarks of PeopleSoft, Inc. All other company and product names may be trademarks of their respective owners. The information contained herein is subject to change without notice.

Open Source Disclosure

This product includes software developed by the Apache Software Foundation (http://www.apache.org/). Copyright (c) 1999-2000 The Apache Software Foundation. All rights reserved. THIS SOFTWARE IS PROVIDED “AS IS’’ AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

PeopleSoft takes no responsibility for its use or distribution of any open source or shareware software or documentation and disclaims any and all liability or damages resulting from use of said software or documentation.

Contents

General Preface

About This PeopleBook lxxxix

PeopleSoft Application Prerequisites lxxxix

PeopleSoft Application Fundamentals lxxxix

Related Documentation xc

Obtaining Documentation Updates xc

Ordering Printed Documentation xc

Typographical Conventions and Visual Cues xci

Typographical Conventions xci

Visual Cues xcii

Comments and Suggestions xciii

Common Elements in These PeopleBooks xciii

Preface

PeopleCode Reference Preface xcv

PeopleCode Reference xcv

Chapter 1

PeopleCode Built-in Functions 1

PeopleCode Typographical Conventions 1

Functions by Category 2

APIs 2

Application Classes 2

Application Engine 2

Arrays 2

Attachment 2

Bulk 3

Business Interlink 3

Chart 3

ChartField 3

Component Buffer 4

Component Interface 5

Conversion 5

Currency and Financial....6
Current Date and Time....6
Custom Display Formats....6
Database and Platform....6
Data Buffer Access....6
Date and Time....7
Defaults, Setting....8
Double-Byte Characters....9
Effective Date and Effective Sequence....9
Email....9
Environment....9
Exceptions....10
Executable Files, Running....10
Files....10
Financial....10
Grids....11
Images....11
Import Manager....11
Integration Broker....11
Internet....11
Java....13
Language Preference and Locale....13
Language Constructs....13
Logical (Tests for Blank Values)....13
Mail....14
Math....14
Menu Appearance....15
Message Catalog and Message Display....15
Messages....15
Mobile....16
Modal Transfers....16
MultiChannel Framework....16
Object-Based....16
Object....18
Page....18
Page Control Appearance....18
Personalizations....19
Process Scheduler....19
Remote Call....19
Saving and Canceling....19

Scroll Select....19
Search Dialog....20
Secondary Pages....20
SQL....20
SQL Date and Time....21
SQL Shortcuts....21
String....22
Subrecords....23
TimeZone....23
Trace Control....23
Transfers....23
Type Checking....24
User Information....24
User Security....24
Validation....25
Workflow....26
XML....26
PeopleCode Built-in Functions and Language Constructs....27
Abs....27
AccruableDays....27
AccrualFactor....28
Acos....29
ActiveRowCount....29
AddAttachment....31
AddEmailAddress....35
AddKeyListItem....36
AddSystemPauseTimes....37
AddToDate....39
AddToDateTime....40
AddToTime....40
All....41
AllOrNone....42
AllowEmplIdChg....43
Amortize....44
Asin....45
Atan....46
BlackScholesCall....46
BlackScholesPut....47
BootstrapYTMs....48
Break....49

BulkDeleteField 49
BulkInsertField 51
BulkModifyPageFieldOrder 54
BulkUpdateIndexes 56
CallAppEngine 58
CancelPubHeaderXmlDoc 61
CancelPubXmlDoc 61
CancelSubXmlDoc 62
ChangeEmailAddress 63
Char 64
CharType 64
ChDir 66
ChDrive 66
CheckMenuItem 67
Clean 67
ClearKeyList 67
ClearSearchDefault 68
ClearSearchEdit 69
Code 69
Codeb 70
CollectGarbage 70
CommitWork 71
CompareLikeFields 73
Component 74
ComponentChanged 75
ContainsCharType 75
ContainsOnlyCharType 77
ConvertChar 79
ConvertCurrency 83
ConvertDatetimeToBase 85
ConvertRate 86
ConvertTimeToBase 87
CopyAttachments 88
CopyFields 89
CopyRow 90
Cos 91
Cot 92
CreateArray 92
CreateArrayAny 93
CreateArrayRept 94

CreateDirectory 96
CreateException 97
CreateJavaArray 98
CreateJavaObject 99
CreateMessage 100
CreateObject 101
CreateObjectArray 103
CreateProcessRequest 104
CreateRecord 105
CreateRowset 107
CreateSOAPDoc 109
CreateSQL 109
CreateXmlDoc 111
CubicSpline 112
CurrEffDt 114
CurrEffRowNum 114
CurrEffSeq 115
CurrentLevelNumber 116
CurrentRowNumber 116
Date 117
Date3 118
DatePart 118
DateTime6 119
DateTimeToHTTP 120
DateTimeToLocalizedString 121
DateTimeToTimeZone 124
DateTimeValue 125
DateValue 126
Day 127
Days 128
Days360 128
Days365 129
DBCSTrim 129
DBPatternMatch 129
Declare Function 130
Decrypt 133
Degrees 134
DeleteAttachment 135
DeleteEmailAddress 137
DeleteImage 138

DeleteRecord 139
DeleteRow 140
DeleteSQL 141
DeleteSystemPauseTimes 143
DeQueue 144
DisableMenuItem 146
DiscardRow 146
DoCancel 147
DoModal 148
DoModalComponent 150
DoModalPanelGroup 154
DoSave 154
DoSaveNow 155
EnableMenuItem 157
EncodeURL 157
EncodeURLForQueryString 158
Encrypt 159
EncryptNodePswd 160
EndMessage 160
EndModal 161
EndModalComponent 162
EnQueue 163
Error 165
EscapeHTML 167
EscapeJavascriptString 168
EscapeWML 169
Evaluate 169
Exact 170
Exec 171
ExecuteRolePeopleCode 173
ExecuteRoleQuery 174
ExecuteRoleWorkflowQuery 175
Exit 176
Exp 176
ExpandBindVar 177
ExpandEnvVar 178
ExpandSqlBinds 178
Fact 179
FetchSQL 180
FetchValue 181

FieldChanged 182
FileExists 184
Find 185
Findb 186
FindCodeSetValues 187
FindFiles 189
FlushBulkInserts 191
For 192
FormatDateTime 192
Forward 193
Function 196
GenerateActGuideContentUrl 198
GenerateActGuidePortalUrl 199
GenerateActGuideRelativeUrl 200
GenerateComponentContentRelURL 201
GenerateComponentContentURL 203
GenerateComponentPortalRelURL 205
GenerateComponentPortalURL 207
GenerateComponentRelativeURL 209
GenerateExternalPortalURL 211
GenerateExternalRelativeURL 212
GenerateHomepagePortalURL 213
GenerateHomepageRelativeURL 214
GenerateMobileTree 215
GenerateQueryContentURL 218
GenerateQueryPortalURL 220
GenerateQueryRelativeURL 221
GenerateScriptContentRelURL 223
GenerateScriptContentURL 224
GenerateScriptPortalRelURL 226
GenerateScriptPortalURL 227
GenerateScriptRelativeURL 229
GenerateTree 230
GenerateWorklistPortalURL 231
GenerateWorklistRelativeURL 232
GetAESection 233
GetAttachment 234
GetBiDoc 236
GetCalendarDate 237
GetChart 239

GetChartURL 240
GetCwd 241
GetEnv 241
GetField 242
GetFile 243
GetGrid 246
GetHTMLText 248
GetImageExtents 249
GetInterlink 250
GetJavaClass 252
GetLevel0 253
GetMethodNames 254
GetMessage 255
GetMessageInstance 255
GetMessageXmlDoc 255
GetNextNumber 257
GetNextNumberWithGaps 259
GetNextProcessInstance 261
GetNRXmlDoc 261
GetPage 262
GetPubContractInstance 263
GetPubHeaderXmlDoc 263
GetPubXmlDoc 264
GetRecord 265
GetRelField 266
GetRow 267
GetRowset 268
GetSelectedTreeNode 269
GetSession 269
GetSetId 270
GetSQL 271
GetStoredFormat 273
GetSubContractInstance 274
GetSubXmlDoc 276
GetSyncLogData 277
GetTreeNodeParent 278
GetTreeNodeRecordName 279
GetTreeNodeValue 279
GetURL 279
GetUserOption 280

GetWLFieldValue 281
Global 281
Gray 282
GrayMenuItem 283
Hash 284
HermiteCubic 285
Hide 285
HideMenuItem 287
HideRow 287
HideScroll 289
HistVolatility 290
Hour 291
Idiv 291
If 292
InitChat 293
InsertImage 295
InsertRow 297
Int 298
Integer 299
IsAlpha 300
IsAlphaNumeric 300
IsDate 301
IsDateTime 302
IsDaylightSavings 302
IsDigits 304
IsHidden 304
IsMenuItemAuthorized 305
IsMessageActive 307
IsModal 307
IsModalComponent 308
IsModalPanelGroup 309
IsNumber 309
IsOperatorInClass 310
IsSearchDialog 310
IsTime 311
IsUserInPermissionList 312
IsUserInRole 312
Left 313
Len 313
Lenb 314

LinearInterp 314
Ln 315
Local 315
Log10 316
Lower 316
LTrim 317
MarkPrimaryEmailAddress 317
MarkWLItemWorked 318
Max 319
MessageBox 319
Min 326
Minute 326
Mod 327
Month 327
MsgGet 328
MsgGetExplainText 329
MsgGetText 330
NextEffDt 331
NextRelEffDt 331
NodeDelete 332
NodeRename 333
NodeSaveAs 333
NodeTranDelete 334
None 335
NotifyQ 336
NumberToString 337
ObjectDoMethod 340
ObjectDoMethodArray 341
ObjectGetProperty 342
ObjectSetProperty 343
OnlyOne 344
OnlyOneOrNone 345
PanelGroupChanged 346
PingNode 346
PriorEffDt 347
PriorRelEffDt 348
PriorValue 349
Product 349
Prompt 350
Proper 352

PublishXmlDoc 352
PutAttachment 353
Quote 356
Radians 356
Rand 357
RecordChanged 358
RecordDeleted 359
RecordNew 361
RefreshTree 363
RelNodeTranDelete 363
RemoteCall 364
RemoveDirectory 367
RenameDBField 368
RenamePage 370
RenameRecord 371
Repeat 372
Replace 373
Rept 373
ReSubmitPubHeaderXmlDoc 374
ReSubmitPubXmlDoc 375
ReSubmitSubXmlDoc 376
Return 376
ReturnToServer 377
ReValidateNRXmlDoc 379
RevalidatePassword 379
Right 381
Round 381
RoundCurrency 382
RowFlush 383
RowScrollSelect 384
RowScrollSelectNew 386
RTrim 387
ScheduleProcess 388
ScrollFlush 388
ScrollSelect 389
ScrollSelectNew 392
Second 393
SendMail 394
SetAuthenticationResult 399
SetChannelStatus 401

SetControlValue 402
SetCursorPos 404
SetDBFieldAuxFlag 405
SetDBFieldCharDefn 407
SetDBFieldFormat 408
SetDBFieldFormatLength 410
SetDBFieldLabel 411
SetDBFieldLength 412
SetDBFieldNotUsed 414
SetDefault 415
SetDefaultAll 416
SetDefaultNext 417
SetDefaultNextRel 417
SetDefaultPrior 418
SetDefaultPriorRel 418
SetDisplayFormat 419
SetLabel 419
SetLanguage 420
SetNextPanel 422
SetNextPage 422
SetPageFieldPageFieldName 423
SetPasswordExpired 424
SetPostReport 425
SetRecFieldEditTable 425
SetRecFieldKey 427
SetReEdit 428
SetSearchDefault 428
SetSearchDialogBehavior 429
SetSearchEdit 430
SetTempTableInstance 431
SetTracePC 432
SetTraceSQL 435
SetupScheduleDefnItem 437
SetUserOption 437
Sign 438
Sin 439
SinglePaymentPV 439
SortScroll 440
Split 441
SQLExec 442

Sqrt 447
StartWork 447
StopFetching 449
StoreSQL 450
String 451
Substitute 452
Substring 452
Substringb 453
SwitchUser 454
SyncRequestXmlDoc 455
Tan 456
Throw 457
Time 458
Time3 458
TimePart 459
TimeToTimeZone 460
TimeValue 461
TimeZoneOffset 461
TotalRowCount 462
Transfer 463
TransferMobilePage 467
TransferNode 468
TransferPortal 471
TransferPanel 475
TransferPage 475
Transform 476
TransformEx 478
TransformExCache 479
TreeDetailInNode 480
TriggerBusinessEvent 481
Truncate 481
Try 482
UnCheckMenuItem 483
Unencode 483
Ungray 484
Unhide 486
UnhideRow 487
UnhideScroll 488
UniformSeriesPV 489
UpdateSysVersion 490

UpdateValue 490
UpdateXmlDoc 492
Upper 492
Value 493
ViewAttachment 494
ViewContentURL 498
ViewURL 498
Warning 499
Weekday 501
While 502
WinEscape 502
WinExec 503
WinMessage 505
Year 509

Chapter 2
Meta-SQL 511
Understanding Meta-SQL 511
Common Terminology Used With Meta-SQL 511
Date Considerations 512
Use of Date, Datetime, or Time Wrappers with an Application Engine Program 512
Date and Time Out Wrappers for Static and Dynamic Views 512
Using {DateTimein-prefix} in SQR 513
Meta-SQL Placement Considerations 513
Meta-SQL Reference 517
%Abs 518
%BINARYSORT 518
%COALESCE 519
%Concat 519
%CurrentDateIn 520
%CurrentDateOut 520
%CurrentDateTimeIn 520
%CurrentDateTimeOut 520
%CurrentTimeIn 521
%CurrentTimeOut 521
%DateAdd 521
%DateDiff 521
%DateIn 522
%DateNull 522

%DateOut 523
%DatePart 523
%DateTimeDiff 523
%DateTimeIn 524
%DateTimeNull 524
%DateTimeOut 525
%DecDiv 525
%DecMult 526
%DTTM 526
%EffDtCheck 527
%FirstRows 528
%InsertSelect 529
%InsertValues 532
%Join 533
%KeyEqual 535
%KeyEqualNoEffDt 536
%Like 537
%LikeExact 539
%List 541
%ListBind 544
%ListEqual 544
%NoUppercase 545
%NumToChar 546
%OldKeyEqual 547
%OPRCLAUSE 547
%Round 548
%SQL 548
%Substring 550
%SUBREC 551
%Table 551
%Test 552
%TextIn 553
%TimeAdd 553
%TimeIn 554
%TimeNull 554
%TimeOut 555
%TimePart 555
%TrimSubstr 556
%Truncate 556
%TruncateTable 557

%UpdatePairs 558
%Upper 559
Meta-SQL Shortcuts 560
%Delete 560
%Insert 560
%SelectAll 560
%SelectDistinct 561
%SelectByKey 562
%SelectByKeyEffDt 562
%Update 562

Chapter 3
System Variables 563
Understanding System Variables 563
%AllowNotification 563
%AllowRecipientLookup 563
%AsOfDate 563
%AuthenticationToken 564
%BPName 564
%ClientDate 564
%ClientTimeZone 564
%Component 564
%CompIntfcName 565
%ContentID 565
%ContentType 565
%Copyright 566
%Currency 566
%Date 566
%DateTime 567
%DbName 567
%DbServerName 567
%DbType 567
%DeviceType 567
%EmailAddress 568
%EmployeeId 568
%ExternalAuthInfo 568
%FilePath 568
%HPTabName 568
%Import 568

%IsMultiLanguageEnabled 569
%Language 569
%Language_Base 569
%LocalNode 569
%Market 570
%MaxInterlinkSize 570
%MaxMessageSize 570
%Menu 571
%MobilePage 571
%Mode 571
%NavigatorHomePermissionList 571
%Node 571
%OperatorClass 572
%OperatorId 572
%OperatorRowLevelSecurityClass 572
%OutDestFormat 572
%OutDestType 573
%Page 573
%Panel 573
%PanelGroup 573
%PasswordExpired 574
%PerfTime 574
%PermissionLists 575
%Portal 575
%PrimaryPermissionList 575
%ProcessProfilePermissionList 576
%PSAuthResult 576
%Request 576
%Response 576
%ResultDocument 576
%Roles 577
%RowSecurityPermissionList 577
%RunningInPortal 577
%ServerTimeZone 577
%Session 577
%SignonUserId 578
%SignOnUserPswd 578
%SMTPBlackberryReplyTo 578
%SMTPGuaranteed 578
%SMTPSender 579

%SQLRows 579
%SyncServer 579
%ThisMobileObject 580
%Time 580
%TransformData 580
%UserDescription 581
%UserId 581
%WLInstanceID 581
%WLName 581

Chapter 4
Meta-HTML 583
Understanding Meta-HTML 583
Variables 584
Functions 584
Comments 584
Alphabetical List of Meta-HTML Elements 585
%Appserver 585
%AppsRel 585
%Browser 585
%BrowserPlatform 585
%BrowserVersion 585
%Cols 586
%Component 586
%Copyright 586
%DBName 586
%DBType 586
%Encode 587
%Formname 587
%Image 587
%JavaScript 587
%LabelTag 588
%Menu 588
%Message 588
%Page 589
%ServicePack 589
%SubmitScriptName 589
%ToolsRel 590
%URL 590

%UserId 590

Chapter 5
AESection Class 591
Understanding AESection Class 591
How an AESection is Accessed 592
AESection Example 593
Data Type of an AESection Object 595
Scope of an AESection Object 595
AESection Class Built-in Function 595
AESection Class Methods 595
AddStep 595
Close 596
Open 597
Save 598
SetSQL 598
SetTemplate 599
AESection Class Property 600
IsOpen 600

Chapter 6
Application Classes 601
Understanding Application Classes 601
When Would You Use Application Classes? 602
Application Classes General Structure 602
Class Name 603
Class Extension 604
Declaration of Public External Interface 604
Declaration of Private Instance Variables and Methods 605
Definition of Methods 606
Constructors 608
External Function Declarations 610
Naming Standards 611
Naming Packages 611
Naming Classes 612
Naming Methods 612
Naming Properties 612
Import Declarations 612

Self-Reference 613
Properties and Constants 615
Differentiating Between Properties and Methods 615
Overriding Properties 616
Using Get and Set with Properties 617
Using Read-Only Properties 617
Using Methods and Properties for Collections 618
Declaring Constants 618
Using Variables in Application Classes 618
Declaring Private Instance Variables 619
Global Variables 619
Overriding Variables and Properties 619
Superclass Reference 619
Downcasting 620
Exception Handling 621
Commenting and Documenting Application Classes 622
Understanding Comments and Documentation 622
Method Header Comments 623
Class Header Comments 624
Designing Base Classes 624
Base Classes 624
Abstract Base Classes 624
Generic Base Classes 625
Utility Classes 625
Declaration of Application Classes 625
Declaring Application Classes 626
Importing Class Names 626
Referencing Superclasses 626
Declaring Private Methods 626
Scope of an Application Class 626
Application Classes Built-in Functions and Language Constructs 626
Class 627
Get 628
Import 629
Method 629
Set 630

Chapter 7
Array Class 633
Understanding Arrays 633
Creating Arrays 633
Populating an Array 635
Removing Items From an Array 636
Creating Empty Arrays 637
Creating and Populating Multi-Dimensional Arrays 637
Using Flattening and Promotion 640
Declaring Array Objects 640
Understanding the Scope of an Array Object 640
Array Class Built-in Functions 641
Array Class Methods 641
Clone 641
Find 642
Get 643
Join 643
Next 644
Pop 645
Push 646
Replace 648
Reverse 649
Set 650
Shift 651
Sort 651
Subarray 653
Substitute 654
Unshift 655
Array Class Properties 656
Dimension 656
Len 656

Chapter 8
Business Interlink Class 659
Understanding Business Interlink Class 659
Using the Interlink Object 660
Deciding Which Methods to Use 660
Executing the Business Interlink Object 660
Supporting Batch Input and Output 661

Supporting Rowsets....661
Using the Flat Table Methods....661
Supporting Dynamic Output....661
Using Hierarchical Data (BIDocs)....662
Using the Incoming Business Interlink Methods and Properties....665
Understanding the State of an Interlink Object....666
Using Business Interlink with Application Engine....666
Using the Data Type of a Business Interlink Object....666
Understanding the Scope of a Business Interlink Object....667
Business Interlink Class Built-in Functions....667
Business Interlink Class Methods....667
AddDoc....667
AddInputRow....669
AddNextDoc....670
AddValue....672
BulkExecute....674
Clear....679
Execute....679
FetchIntoRecord....681
FetchIntoRowset....683
FetchNextRow....686
GetCount....687
GetDoc....689
GetFieldCount....691
GetFieldType....692
GetFieldValue....693
GetInputDocs....694
GetNextDoc....695
GetOutputDocs....697
GetPreviousDoc....698
GetStatus....699
GetValue....700
InputRowset....702
MoveFirst....704
MoveNext....705
MoveToDoc....706
ResetCursor....707
Incoming Business Interlink Methods....708
AddAttribute....708
AddComment....709

AddProcessInstruction....710
AddText....711
CreateElement....712
GenXMLString....713
GetAttributeName....714
GetAttributeValue....715
GetNode....715
ParseXMLString....717
Incoming Business Interlink Properties....717
AttributeCount....718
ChildNodeCount....718
NodeName....719
NodeType....720
NodeValue....721
Business Interlink Class Property....721
StopAtError....721
Configuration Parameters....722
Interlink Plug-in Configuration Parameters....722
URL Configuration Parameter....723
BIDocValidate Configuration Parameter....723

Chapter 9
Chart Class....725
Understanding the Chart Class....725
Chart Terms....727
Chart Types....728
2D Bar....728
2D Histogram....729
2D Horizontal Bar....729
2D Horizontal Stacked Bar....730
2D Horizontal Percent Bar....730
2D Line....731
2D Percent Bar....731
2D Pie....732
2D Stacked Bar....732
3D Bar....733
3D Percent Bar....733
3D Pie....734
3D Stacked Bar....734

Style Sheets and Charts 735
Considerations Using Style Sheets 735
Default Style Sheet Classes 736
Component Processor Considerations 736
Translation Considerations 737
iScripts and Charts 737
Error Handling 737
Data Type of a Chart Object 738
Scope of a Chart Object 738
Chart Class Reference 738
Chart Class Built-in Functions 738
Chart Data Methods 739
Refresh 739
SetData 739
SetDataSeries 741
SetDataHints 741
SetDataURLs 744
SetDataXAxis 745
SetDataYAxis 746
Chart Data Properties 747
DataStartRow 747
DataWidth 747
ImageMap 748
Chart Appearance Properties 748
GridLines 748
Height 748
IsDrillable 749
IsPlainImage 749
LineType 750
RotationAngle 750
Style 750
StyleSheet 751
Type 751
Width 752
Chart Colors 752
Chart Color Methods 753
SetColorArray 753
SetDataColor 754
Chart Axes Methods 755
SetXAxisLabels 755

SetYAxisLabels....755
Chart Axes Properties....756
XAxisCross....756
XAxisLabelOrient....757
XAxisStyle....757
YAxisLabelOrient....758
YAxisMax....758
YAxisMin....758
YAxisStyle....758
YAxisTicks....759
Chart Title Properties....759
MainTitle....759
MainTitleOrient....759
MainTitleStyle....759
XAxisTitle....760
XAxisTitleOrient....760
XAxisTitleStyle....760
YAxisTitle....760
YAxisTitleOrient....760
YAxisTitleStyle....761
Chart Legend Methods....761
SetLegend....761
Chart Legend Properties....762
HasLegend....762
LegendMaxEntries....762
LegendPosition....763
LegendStyle....763
Chart Overlay Methods....763
SetOLData....763
SetOLDataSeries....764
SetOLDataXAxis....764
SetOLDataYAxis....765
Chart Overlay Properties....765
OLLineType....765
OLType....766
Chart Examples....766
Creating a Chart....766
Creating a 'Gantt' Chart....768
Creating a Chart Using an iScript....770

Chapter 10
Component Interface Classes 773
Understanding Component Interface Class 773
Life Cycle of a Component Interface 774
Setting Component Interface Keys 776
Standard and User-Defined Component Interface Methods 776
Accessing Component Interface Standard Properties 777
Accessing User-Defined Component Interface Properties 778
Data Type of a Component Interface Object 778
Scope of a Component Interface Object 779
Implementing a Component Interface 779
Component Interface Methods and Timeouts 780
Traversing a Component Interface and Using Data Collections 780
Working With Effective-Dated Data 782
How a Developer Uses GetEffectiveItem 782
Reusing Existing Code 783
Differences in Search Dialog Processing 783
Differences in PeopleCode Event and Function Behavior 783
Using %Menu Conditions 784
Session Class Methods 784
FindCompIntfcs 784
GetCompIntfc 785
Component Interface Collection Methods 785
First 786
Item 786
Next 786
Component Interface Collection Property 787
Count 787
Component Interface Class Methods 787
Cancel 787
CopyRowset 787
CopyRowsetDelta 790
CopySetupRowset 792
CopySetupRowsetDelta 794
Create 796
Find 797
Get 798
GetPropertyByName 798
Save 800
SetPropertyByName 800

Component Interface Class Properties 801
ComponentName 801
CreateKeyInfoCollection 802
EditHistoryItems 802
FindKeyInfoCollection 802
GetDummyRows 803
GetHistoryItems 803
GetKeyInfoCollection 804
InteractiveMode 804
PropertyInfoCollection 804
StopOnFirstError 804
Data Collection Methods 805
CurrentItem 805
DeleteItem 805
GetEffectiveItem 807
GetEffectiveItemNum 807
InsertItem 808
Item 809
ItemByKeys 809
Data Collection Properties 811
Count 811
CurrentItemNumber 811
Data Item Class Property 811
ItemNum 812
Accessing the Structure of a Component Interface 812
CompIntfPropInfoCollection Collection 812
CompIntfPropInfoCollection Collection Methods 814
First 814
Item 815
Next 815
CompIntfPropInfoCollection Collection Properties 815
Count 816
CompIntfPropInfoCollection Object Properties 816
Format 816
IsCollection 817
IsReadOnly 817
Key 817
LabelLong 817
LabelShort 817
Length 818

Name....818
Prompt....818
PropertyInfoCollection....818
Required....819
Type....819
Xlat....819
YesNo....820
Component Interface Examples....820
Create a New Instance of Data Example....820
Getting an Existing Instance of Data Example....822
Retrieving a List of Instance of Data Example....825
Inserting Effective-Dated Data Example....827
Inserting Effective-Dated Data Example Using Visual Basic....829
Inserting or Deleting a Row of Data Example....831

Chapter 11
Exception Class....835
Understanding Exception Class....835
How Exceptions Are Thrown....836
When to Use Exceptions....836
Try-Catch Sections....837
Data Type for Exception Class Objects....838
Scope of Exception Class Objects....838
Exception Class Built-in Functions....838
Exception Class Methods....839
GetSubstitution....839
Output....839
SetSubstitution....840
ToString....840
Exception Class Properties....841
Context....841
DefaultText....841
MessageNumber....842
MessageSetNumber....842
MessageSeverity....842
SubstitutionCount....842

Chapter 12
Field Class 843
Understanding the Field Class 843
Shortcut Considerations 843
Data Type for a Field Object 844
Scope of a Field Object 844
Field Class Built-in Function 844
Field Class Methods 844
AddDropDownItem 845
ClearDropDownList 846
GetAuxFlag 846
GetLongLabel 847
GetRelated 848
GetShortLabel 849
SearchClear 850
SetCursorPos 851
SetDefault 851
Field Class Properties 852
DataAreaCollapsed 852
DisplayFormat 853
DisplayOnly 854
EditError 854
Enabled 855
FieldLength 855
FormatLength 855
FormattedValue 855
IsAltKey 856
IsAuditFieldAdd 856
IsAuditFieldChg 857
IsAuditFieldDel 857
IsAutoUpdate 857
IsChanged 857
IsDateRangeEdit 857
IsDescKey 858
IsDuplKey 858
IsEditTable 858
IsEditXlat 858
IsFromSearchField 858
IsInBuf 858
IsKey 859

IsListItem....859
IsNotUsed....859
IsRequired....860
IsSearchItem....860
IsSystem....860
IsThroughSearchField....860
IsUseDefaultLabel....860
IsYesNo....861
Label....861
LabelImage....862
LongTranslateValue....862
MessageNumber....863
MessageSetNumber....863
Name....864
OriginalValue....864
ParentRecord....864
PromptTableName....865
SearchDefault....865
SearchEdit....865
ShortTranslateValue....866
ShowRequiredFieldCue....867
SqlText....867
StoredFormat....868
Style....868
Type....869
Value....869
Visible....870

Chapter 13
File Class 873
Understanding File Layout....873
Data Type of a File Object....874
Scope of a File Object 874
File Security Considerations 875
File Access Interruption Recovery 875
Plain Text Files 876
Automatic PeopleCode Generation....877
File Layout Error Processing 878
File Class Built-in Functions....879

File Class Methods 879
Close 879
CreateRowset 880
Delete 881
GetPosition 882
Open 882
ReadLine 885
ReadRowset 886
SetFileId 888
SetFileLayout 890
SetPosition 890
SetRecTerminator 891
WriteLine 892
WriteRaw 893
WriteRecord 894
WriteRowset 896
WriteString 897
File Class Properties 898
CurrentRecord 898
IgnoreInvalidId 899
IsError 899
IsNewFileId 900
IsOpen 900
Name 901
File Layout Examples 901
WriteRecord Example 902
ReadRecord Example 905
WriteRowset Example 906
ReadRowset Example 910
File Rowset Considerations 911
Application Engine Example 911
Multiple File Layouts 915
Reading Multiple File Layouts 915
Writing Multiple File Layouts 917

Chapter 14
Grid Classes 921
Understanding Grid and GridColumn Classes 921
Shortcut Considerations 922

The Grid Class in PeopleCode 922
Data Type for a Grid or Grid Column Object 923
Scope of a Grid or Grid Column Object 923
Grid Class Built-in Function 923
Grid Class Method 924
GetColumn 924
Grid Class Properties 926
gridcolumn 926
Label 926
GridColumn Class 926
GridColumn Class Properties 927
Enabled 927
Label 927
Name 928
Visible 928

Chapter 15
Internet Script Classes (iScript) 931
Understanding Internet Script Classes 931
URL vs. URI 932
Web Libraries 933
iScript Security 934
When to Use an iScript 935
Style Sheets and Styles 936
Other Considerations 936
Details of an iScript URL 936
The Generate Functions 937
Error Handling 939
Scope of the Internet Script Classes 939
Data Types of the iScript Classes 939
Internet Script Classes 940
Request Class 940
Response 940
Cookies 941
Internet Script Classes Built-in Functions 941
Request Class Methods 942
GetContentBody 942
GetCookieNames 942
GetCookieValue 943

GetHeader....943
GetHeaderNames....943
GetHelpURL....943
GetParameter....944
GetParameterNames....944
GetParameterValues....944
Request Class Properties....944
AuthTokenDomain....944
ByPassSignOn....945
BrowserPlatform....945
BrowserType....945
BrowserVersion....945
ContentURI....946
ExpireMeta....946
FullURI....946
HTTPMethod....947
LogoutURL....947
PathInfo....947
Protocol....947
QueryString....948
RelativeURL....948
RemoteAddr....948
RemoteHost....948
RequestURI....948
Scheme....949
ServerName....949
ServerPort....950
Timeout....950
Response Class....950
Response Class Methods....950
Clear....950
CreateCookie....951
GetCookie....951
GetCookieNames....951
GetHeader....951
GetHeaderNames....951
GetImageURL....952
GetJavaScriptURL....952
GetStyleSheetURL....953
RedirectURL....953

SetContentType....954
SetHeader....954
UseSimpleURL....954
Write....955
WriteLine....955
Response Class Property....956
Charset....956
Cookie Class....956
Cookie Class Properties....956
Domain....956
MaxAge....957
Name....957
Path....957
Secure....957
Value....957

Chapter 16
Java Class....959
Understanding Java Class....959
Supported Versions of Java....960
Naming Standards for Classes and Packages....960
System Setup for Java Classes....960
From PeopleCode to Java....961
State Management Concerns....962
CreateJavaObject Example....963
CreateJavaArray Example....963
GetJavaClass Example....964
From Java to PeopleCode....964
SysVar Java Class....964
SysCon Java Class....965
Func Java Class....965
Name Java Class....965
Using PeopleCode Objects....965
Using Application Classes From Java to PeopleCode....967
PeopleCode and Java Data Types Mapping....968
Considerations When Using the PeopleCode Java Functions....969
Error Handling and the PeopleCode Java Functions....970
The Java Debugging Environment....970
Data Type of a Java Object....971

Scope of a Java Object....971
PeopleCode Java Built-in Functions....971

Chapter 17
Message Class....973
Understanding Message Class....973
Messaging PeopleCode Events....974
Data Type of a Message Object....974
Scope of a Message Object....974
Message Object Population....975
Considerations When Populating a Rowset From a Message....976
Considerations for Publishing and Subscribing to Partial Records....978
Considerations When Subscribing to Character Fields....978
Error Handling....979
Message Class Built-In Functions....980
Message Class Methods....980
Clone....980
CopyRowset....981
CopyRowsetDelta....982
CopyRowsetDeltaOriginal....984
ExecuteEdits....987
GenXMLString....989
GetQueryString....990
GetRowset....991
LoadXMLString....992
Publish....992
SetEditTable....994
SetQueryString....995
SetStatus....996
SyncRequest....997
Update....998
Message Class Properties....998
ChannelName....998
Cookies....999
DefaultMessageVersion....1000
DoNotPubToNodeName....1000
GUID....1000
IsActive....1001
IsDelta....1001

IsEditError....1002
IsEmpty....1003
IsLocal....1003
MessageDetail....1003
Name....1004
NRId....1004
PubID....1004
PubNodeName....1004
ResponseStatus....1005
Size....1005
SubName....1006
SubscriptionProcessId....1006

Chapter 18
Mobile Classes 1009
Mobile Classes Overview....1009
Understanding the Mobile Device and the Component Interface....1010
Mobile Class Hierarchy....1013
HTML Areas Used in Mobile....1016
Debugging in Mobile....1017
Debug Settings....1018
Trace Settings....1019
Error Handling....1019
Data Types for Mobile....1020
Scope of Mobile Objects....1020
Variables in Mobile PeopleCode....1020
Mobile Functions....1020
Session Class Method....1021
GetCompIntfc....1021
Mobile Class....1022
Mobile Class Methods....1022
Create....1022
Find....1023
Get....1024
GetName....1024
GetParent....1025
GetPeerDefaultAttributesByName....1025
GetPropertyAttrsByName....1026
GetPropertyByName....1027

GetPropertyInfoByName....1027
GetTopParent....1028
IsNew....1029
IsNewAndNeverSaved....1029
IsSameAs....1030
ReadBlobFromFile....1030
Save....1031
SetModified....1032
SetPropertyByName....1033
ValidateEnum....1034
ValuesDifferFromLastSync....1034
WasSaved....1035
WriteBlobToFile....1035
Mobile Class Property....1036
PropertyInfoCollection....1037
PropertyInfoCollection....1037
PropertyInfoCollection Method....1037
Item....1037
PropertyInfoCollection Property....1038
Count....1038
PropertyInfo Class....1038
PropertyInfo Class Properties....1039
DefaultValue....1039
HasDefaultValue....1039
IsCollection....1039
IsRefToParent....1040
IsPeerReference....1040
IsSimpleApplicationProperty....1040
IsSystemProperty....1041
Name....1041
PropertyInfoCollection....1042
CI Collection Class....1042
CI Collection Method....1042
Item....1042
CI Collection Properties....1043
Count....1043
DataCollection Class....1043
Data Collection Methods....1044
GetListViewAttrs....1044
Item....1045

DeleteItem....1046
InsertItem....1047
Data Collection Properties....1048
Count....1048
ListViewAttrs Class....1049
ListViewAttrs Method....1049
IsSet....1049
ListViewAttrs Properties....1050
DetailViewLabel....1050
Label....1050
ListViewLabel....1051
Visible....1051
PropertyAttrs Class....1052
PropertyAttrs Method....1052
IsSet....1052
PropertyAttrs Properties....1053
DisplayOnly....1053
InError....1054
Label....1054
ShowRequiredCue....1055
Visible....1055
PeerDefaultAttributes Class....1056
PeerDefaultAttributes Class Method....1056
IsSet....1056
PeerDefaultAttributes Properties....1057
DisplayOnly....1057
Label....1057
Visible....1057
Classes and Functions and System Variables Used in Mobile....1057
PeopleCode Classes Used in Mobile....1057
System Variables Used in Mobile....1058
Built-In Functions Used in Mobile....1059

Chapter 19
Notification Classes....1061
Understanding Notification Classes....1061
Scope of the Notification Classes....1061
Data Types of the Notification Classes....1062
How to Import the Notification Classes....1062

How to Create a Notification Object 1062
Notification Classes Constructors 1063
Notification 1063
NotificationAddress 1064
NotificationTemplate 1064
WorklistEntry 1065
Notification Class 1066
Notification Class Import Statements 1066
Notification Class Method 1066
Send 1067
Notification Class Properties 1067
ContentType 1067
dttmCreated 1067
EmailReplyTo 1068
FileNames 1068
FileTitles 1068
language_cd 1068
Message 1068
NotifyBCC 1069
NotifyCC 1069
NotifyFrom 1069
NotifyGuid 1069
NotifyTo 1070
SourceComponent 1070
SourceMarket 1070
SourceMenu 1070
Subject 1070
Template 1071
NotificationAddress Class 1071
NotificationAddress Class Properties 1071
Channel 1071
Description 1071
EmailId 1072
Language 1072
Oprid 1072
NotificationTemplate Class 1072
NotificationTemplate Class Methods 1072
GetAndExpandTemplate 1072
SetupCompVarsAndRcpts 1073
SetupGenericVars 1074

NotificationTemplate Class Properties 1075
ComponentId 1075
Instruction 1075
Language 1075
Market 1076
Priority 1076
Responses 1076
Subject 1076
TemplateId 1076
TemplateType 1076
Text 1077
WorklistEntry Class 1077
WorklistEntry Methods 1077
Create 1077
Save 1078
Update 1079
WorklistEntry Properties 1080
actiondttm 1080
busactivity 1080
buseventname 1080
busprocname 1080
commentshort 1080
do_replicate_flag 1080
instanceid 1081
instselecteddttm 1081
inststatus 1081
insttimeoutddttm 1082
instworkeddttm 1082
oprid 1082
originatorid 1082
prevoprid 1082
syncid 1082
timedout 1082
transactionid 1083
url 1083
wl_priority 1083
wldaystoselect 1083
wldaystowork 1084
worklistname 1084
Notification Classes Examples 1084

Creating a WorklistEntry 1084
Updating a WorklistEntry 1086

Chapter 20
Page Class 1089
Understanding Page Class....1089
Shortcut Considerations 1089
Data Type of a Page Object 1090
Scope of a Page Object 1090
Page Class Built-in Function....1090
Page Class Properties 1090
DisplayOnly....1090
Name....1091
Visible....1091

Chapter 21
PortalRegistry Classes 1093
Understanding Portal Registry 1093
Folders....1094
Content References....1094
Nodes....1095
Security....1095
Attributes....1096
Additional Portal Objects....1096
Using the PortalRegistry API 1096
Using Security 1097
Using Object Properties....1097
Accessing Objects....1097
Using PermissionValue and Cascading Permissions 1098
Working With ValidFrom and ValidTo 1100
Doing Error Handling 1100
Understanding the Life-Cycle of a PortalRegistry Object 1101
Viewing the PortalRegistry Class Hierarchy 1104
Using Content References 1106
UsageType....1107
TemplateType....1109
URLType....1109
Nodes and URL....1110

Naming Conventions....1111
Deleting Content Considerations....1111
Saving Content Considerations....1112
Data Type of a PortalRegistry Object....1112
Scope of a PortalRegistry Object....1113
PortalRegistry Reference....1113
Session Class Methods....1113
FindPortalRegistries....1114
GetLocalNode....1114
GetNodes....1115
GetPortalRegistry....1115
GetRemoteNodes....1116
PortalRegistry Class....1116
PortalRegistry Class Methods....1117
Close....1117
CopyObject....1117
Create....1119
CreateRemote....1119
Delete....1120
DeleteHomepage....1121
FindCRefByName....1121
FindCRefByURL....1122
FindFolderByName....1123
FindPgltByName....1124
GetAbsoluteContentURL....1125
GetDefaultHPTabOID....1126
GetQualifiedURL....1126
GrantPermissionForComponent....1126
GrantPermissionForScript....1127
Open....1128
PermissionListDelete....1129
PermissionListSaveAs....1129
RevokePermissionForComponent....1130
RevokePermissionForScript....1131
Save....1131
PortalRegistry Class Properties....1132
DefaultTemplate....1132
Description....1133
Favorites....1133
Homepage....1133

Name....1133
NodeTemplates....1133
OwnerId....1134
PageletCategories....1134
Portals....1134
RootFolder....1134
TabDefinitions....1134
TemplateObject....1135
PortalRegistry Collection....1135
PortalRegistry Collection Methods....1135
First....1135
Item....1135
Next....1136
PortalRegistry Collection Property....1136
Count....1136
Node Class....1137
Node Class Properties....1137
ActiveNode....1137
AppsRelease....1137
ContentURI....1137
DefaultPortalName....1138
Description....1138
IsDefault....1138
IsLocal....1138
Name....1138
NodePassword....1139
NodeType....1139
PortalURI....1139
ToolsRelease....1139
Node Collection....1140
Node Collection Methods....1140
First....1140
ItemByName....1140
Next....1141
Node Collection Property....1141
Count....1141
RemoteNode Collection....1142
RemoteNode Collection Methods....1142
First....1142
ItemByName....1142

Next....1143
RemoteNode Collection Property....1143
Count....1143
Portal Class....1144
Portal Class Method....1144
Save....1144
Portal Class Properties....1144
HostNodeName....1144
IsLocal....1145
Name....1145
Portal Collection....1145
Portal Collection Methods....1145
First....1145
ItemByName....1146
Next....1146
Portal Collection Properties....1147
Count....1147
NodeTemplate Class....1147
NodeTemplate Class Properties....1147
DefaultTemplate....1147
Name....1147
TemplateObject....1148
NodeTemplate Collection....1148
NodeTemplate Collection Methods....1148
DeleteItem....1148
First....1149
InsertItem....1149
ItemByName....1150
Next....1150
NodeTemplate Collection Properties....1150
Count....1151
Folder Class....1151
Folder Class Method....1151
Save....1151
Folder Class Properties....1152
Attributes....1152
Author....1152
AuthorAccess....1152
Authorized....1152
CascadedPermissions....1153

ContentRefs....1153
CreationDate....1153
Description....1154
Folders....1154
IsMobile....1154
IsVisible....1154
Label....1155
Name....1155
OwnerId....1155
ParentName....1155
Path....1155
Permissions....1156
Product....1156
SequenceNumber....1156
PublicAccess....1156
ValidFrom....1157
ValidTo....1157
Folder Collection....1157
Folder Collection Methods....1157
DeleteItem....1157
First....1158
InsertItem....1158
ItemByName....1159
Next....1159
Folder Collection Property....1160
Count....1160
Content Reference Class....1160
Content Reference Class Methods....1161
Save....1161
Content Reference Class Properties....1162
AbsoluteContentURL....1162
AbsolutePortalURL....1162
AssignedPagelets....1162
Attributes....1162
Author....1163
AuthorAccess....1163
Authorized....1163
CascadedPermissions....1163
ContentProvider....1164
CreationDate....1164

Data....1164
Description....1164
HtmlText....1165
IsMobile....1165
IsVisible....1165
Label....1165
Name....1166
OwnerId....1166
ParentName....1166
Path....1166
Permissions....1166
Product....1167
PublicAccess....1167
QualifiedURL....1167
RelativeURL....1167
SequenceNumber....1168
StorageType....1168
Template....1169
TemplateObject....1169
TemplateType....1170
URL....1170
URLType....1171
UsageType....1171
ValidFrom....1172
ValidTo....1172
Content Reference Collection....1172
Content Reference Collection Methods....1172
DeleteItem....1173
First....1173
InsertItem....1174
ItemByName....1175
Next....1175
Content Reference Collection Property....1175
Count....1176
AttributeValue Class....1176
AttributeValue Class Properties....1176
Label....1176
Name....1176
Translatable....1177
Value....1177

Attribute Collection....1178
Attribute Collection Methods....1178
DeleteItem....1178
First....1179
InsertItem....1179
ItemByName....1180
Next....1180
Attribute Collection Property....1181
Count....1181
PermissionValue Class....1181
PermissionValue Class Properties....1181
Cascade....1182
Name....1182
PermissionValue Collection....1182
PermissionValue Collection Methods....1183
DeleteItem....1183
First....1183
InsertItem....1184
ItemByName....1184
Next....1185
PermissionValue Collection Property....1185
Count....1185
TabDefinition Class....1186
TabDefinition Class Methods....1186
Save....1186
TabDefinition Class Properties....1187
AssignedPagelets....1187
AvailableCategories....1187
AvailablePagelets....1187
Attributes....1187
Author....1188
AuthorAccess....1188
ColumnLayout....1188
CreationDate....1188
Description....1188
DynamicCategories....1189
HelpID....1189
HtmlText....1189
IsHideActionBar....1189
IsLayoutLocked....1189

IsRenameable....1189
Label....1190
LayoutBehavior....1190
Name....1190
OwnerId....1190
Product....1191
PublicAccess....1191
QualifiedURL....1191
SequenceNumber....1191
StyleSheet....1191
ValidFrom....1192
ValidTo....1192
TabDefinition Collection....1192
TabDefinition Collection Methods....1192
DeleteItem....1192
First....1193
InsertItem....1193
ItemByName....1194
Next....1194
TabDefinition Collection Property....1195
Count....1195
AssignedPagelet Class....1195
AssignedPagelet Class Properties....1195
Column....1196
LayoutBehavior....1196
PageletName....1196
Row....1196
AssignedPagelet Collection....1197
AssignedPagelet Collection Methods....1197
DeleteItem....1197
First....1197
InsertItem....1198
ItemByName....1199
Next....1199
AssignedPagelet Collection Property....1200
Count....1200
AvailableCategory Class....1200
AvailableCategory Class Properties....1200
AvailablePagelets....1200
CategoryName....1201

AvailableCategory Collection....1201
AvailableCategory Collection Methods....1201
First....1201
ItemByName....1201
Next....1202
AvailableCategory Collection Property....1202
Count....1202
AvailablePagelet Class....1203
AvailablePagelet Class Properties....1203
CategoryLabel....1203
CategoryName....1203
Column....1203
LayoutBehavior....1204
PageletLabel....1204
PageletName....1204
Row....1204
AvailablePagelet Collection....1205
AvailablePagelet Collection Methods....1205
First....1205
ItemByName....1205
Next....1206
AvailablePagelet Collection Property....1206
Count....1206
DynamicCategory Collection....1207
DynamicCategory Collection Methods....1207
DeleteItem....1207
First....1208
InsertItem....1208
ItemByName....1209
Next....1209
DynamicCategory Collection Property....1209
Count....1210
PageletCategory Class....1210
PageletCategory Class Method....1210
Save....1210
PageletCategory Class Properties....1211
Attributes....1211
Author....1211
AuthorAccess....1211
Authorized....1211

CascadedPermissions....1212
CreationDate....1212
Description....1212
Label....1212
Name....1213
OwnerId....1213
Pagelets....1213
Permissions....1213
Product....1214
PublicAccess....1214
SequenceNumber....1214
PageletCategory Collection....1214
PageletCategory Collection Methods....1214
DeleteItem....1215
First....1215
InsertItem....1216
ItemByName....1216
Next....1217
PageletCategory Collection Property....1217
Count....1217
Pagelet Class....1217
Pagelet Class Method....1218
Save....1218
Pagelet Class Properties....1218
Attributes....1218
Author....1219
AuthorAccess....1219
Authorized....1219
CascadedPermissions....1219
ContentProvider....1219
CreationDate....1220
DefaultColumn....1220
Description....1220
EditPageContentProvider....1220
EditPageQueryString....1220
HelpID....1221
IsHideMinimize....1221
Label....1221
Name....1221
OwnerId....1221

ParentName....1221
Permissions....1222
Product....1222
PublicAccess....1222
QualifiedURL....1222
SequenceNumber....1223
URL....1223
URLType....1223
Pagelet Collection....1224
Pagelet Collection Methods....1224
DeleteItem....1224
First....1225
InsertItem....1225
ItemByName....1226
Next....1226
Pagelet Collection Property....1227
Count....1227
UserHomepage Class....1227
UserHomepage Class Method....1227
Save....1227
UserHomepage Class Properties....1228
Greeting....1228
UserId....1228
UserTab....1228
UserTab Class....1229
UserTab Class Properties....1229
ColumnLayout....1229
Label....1229
QualifiedURL....1229
SelectedPagelets....1230
SequenceNumber....1230
TabName....1230
UserTab Collection....1230
UserTab Collection Methods....1230
DeleteItem....1230
First....1231
InsertItem....1231
ItemByName....1232
Next....1232
UserTab Collection Property....1233

Count....1233
SelectedPagelet Class....1233
SelectedPagelet Class Properties....1233
CategoryName....1233
Column....1234
IsMinimized....1234
PageletName....1234
Row....1234
SelectedPagelet Collection....1234
SelectedPagelet Collection Methods....1235
DeleteItem....1235
First....1235
InsertItem....1236
ItemByName....1236
Next....1237
SelectedPagelet Collection Property....1237
Count....1237
Favorite Class....1237
Favorite Class Properties....1238
CRefName....1238
Label....1238
QualifiedURL....1238
SequenceNumber....1238
URL....1238
Favorite Collection....1239
Favorite Collection Methods....1239
DeleteItem....1239
First....1240
InsertItem....1240
ItemByLabel....1240
Next....1241
Save....1241
Favorite Collection Property....1242
Count....1242
PortalRegistry Classes Example....1242
Changing PortalRegistry Properties....1242
Adding a ContentProvider....1243
Changing ContentProvider Properties....1245
Adding a Folder....1248
Adding a Content Reference....1250

Setting Permissions Using the PermissionValue Object....1253
Using Attributes....1255
Visual Basic Example....1256
C/C++ Example....1256

Chapter 22
PostReport Class....1261
PostReport Class Overview....1261
Determining the Distribution List....1261
Data Type of a PostReport Object....1262
Scope of a PostReport Object....1262
PostReport Class Built-in Function....1262
PostReport Class Methods....1262
AddDistributionOption....1262
Put....1263
PostReport Class Properties....1264
ExpirationDate....1264
ReportFolder....1264
ReportId....1264
OutDestFormat....1265
ProcessInstance....1265
ProcessName....1265
ProcessType....1266
ReportDescr....1266
ReportFolder....1266
ServerName....1267
PostReport Class Examples....1267

Chapter 23
ProcessRequest Class....1269
Understanding ProcessRequest Class....1269
Data Type of a ProcessRequest Object....1270
Scope of a ProcessRequest Object....1270
Options for Items In Jobs and Jobsets....1270
Values for Output Type and Format....1274
Alternative Option To Specify Email or Web Attributes....1276
ProcessRequest Class Built-in Functions....1277
ProcessRequest Class Methods....1278

AddDistributionOption....1278
PrintJobHTMLRpt....1279
PrintJobRqstRpt....1280
PrintSchdlHTMLRpt....1283
RunJobSetNow....1286
Schedule....1286
SetEmailOption....1287
SetItemFolder....1288
SetOutputOption....1289
UpdateRunStatus....1290
ProcessRequest Class Properties....1291
EmailAttachLog....1291
EmailSubject....1291
EmailText....1291
EmailWebReport....1292
JobName....1292
LanguageCd....1292
NotifyTextMsgNum....1292
NotifyTextMsgSet....1293
OutDest....1293
OutDestFormat....1294
OutDestType....1295
PortalFolder....1295
ProcessInstance....1295
ProcessName....1296
ProcessType....1296
RunControlID....1297
RunDateTime....1297
RunLocation....1298
RunRecurrence....1298
RunStatus....1298
Status....1299
TimeZone....1300
ProcessRequest Class Examples....1300
Scheduling a Single Process....1300
Scheduling a Job Where Job Item Changes Are Not Made....1301
Scheduling a Job Where Job Item Changes Are Made....1302

Chapter 24
Query Classes 1303
Understanding Query Classes 1303
Collections in the Query Classes 1303
Life-Cycle of a Query 1304
Query Class Hierarchy 1305
Query Classes Overview 1307
Query 1308
QuerySelect 1308
QueryRecords 1308
QueryOutputFields 1308
QuerySelectedFields 1308
QueryCriteria 1308
QueryDBRecords and QueryDBRecordFields 1309
Working With Criteria and Expressions 1309
Setting the Type 1309
Considerations Adding New Expressions 1310
Operator and Expression 2 Dependencies 1310
Query Monitor 1311
Using Query Metadata 1311
Running a Query 1313
Specifying User's Language 1314
Query Security 1314
Error Handling With Query Classes 1314
Understanding QueryOutputFields and QuerySelectedFields 1315
Understanding Having Criteria 1316
Data Type of Query Objects 1316
Scope of the Query Objects 1316
Session Class Methods 1317
FindQueryDBRecords 1317
FindQueries 1318
FindQueriesDateRange 1318
GetQuery 1319
GetQuerySecurityProfile 1320
SearchQueryDBRecords 1320
SearchPrivateQueries 1321
SearchPublicQueries 1323
Query Collection 1324
Query Collection Methods 1324
First 1324

Item....1325
ItemByName....1325
Next....1326
Query Collection Property....1326
Count....1326
Query Class....1327
Query Class Methods....1327
AddQuerySelect....1327
AddPrompt....1327
Close....1328
Create....1329
Delete....1330
DeletePrompt....1330
FindExpression....1331
FormatResultString....1331
Open....1332
Rename....1333
RunToFile....1334
RunToRowset....1336
Save....1338
Query Class Properties....1339
Approved....1339
ApproveOprId....1339
ApproveUserId....1340
ApproveDtTm....1340
CreateOprId....1340
CreateDtTm....1340
CreateUserId....1341
Description....1341
ExecAppName....1341
ExecLogging....1341
LastSQLErrorCode....1342
LastUpdDttm....1342
LastUpdOprId....1342
LongDescription....1342
Metadata....1342
Name....1343
PDFFont....1343
Prompts....1343
PromptRecord....1344

PublicPrivate....1344
QuerySelect....1344
QueryStatistics....1345
RunTimePrompts....1345
SQL....1345
Type....1345
QuerySelect Class....1346
QuerySelect Methods....1347
AddAllFields....1347
AddCriteria....1347
AddExpression....1348
AddHavingCriteria....1349
AddQueryExpressionField....1349
AddQueryOutputField....1350
AddQueryRecord....1350
AddQuerySelectedField....1351
DeleteCriteria....1351
DeleteExpression....1352
DeleteField....1352
DeleteHavingCriteria....1353
DeleteRecord....1353
QuerySelect Properties....1354
Criteria....1354
Distinct....1354
Expressions....1354
HavingCriteria....1355
ParentSelectNum....1355
QueryOutputFields....1355
QueryRecords....1355
QuerySelectedFields....1356
QuerySelects....1356
SelectNum....1356
SelectType....1356
QuerySelect Collection....1357
QuerySelect Collection Methods....1357
AddUnion....1357
DeleteQuerySelect....1357
First....1358
Item....1358
ItemBySelNum....1359

Next....1359
QuerySelect Collection Property....1360
Count....1360
QueryRecord Collection....1360
QueryRecord Collection Methods....1360
First....1360
Item....1361
ItemByAlias....1361
Next....1362
QueryRecord Collection Property....1362
Count....1362
QueryRecord Class....1363
QueryRecord Class Method....1363
GetField....1363
QueryRecord Class Properties....1364
Description....1364
JoinAlias....1364
JoinFieldName....1364
JoinType....1364
Name....1365
QueryFields....1365
RecordAlias....1365
QueryField Collection....1365
QueryField Collection Methods....1366
First....1366
Item....1366
ItemByNameAndAlias....1367
ItemByExpNum....1367
Next....1368
QueryField Collection Property....1368
Count....1368
QueryField Class....1368
QueryField Methods....1369
AddTranslateExpression....1369
AddTranslateField....1369
QueryField Class Properties....1370
Aggregate....1370
ColumnNumber....1370
Description....1370
Decimal....1371

ExpNum....1371
Flag....1371
Format....1371
HeadingText....1372
HeadingType....1372
HeadingUniqueFieldName....1372
Length....1373
LongName....1373
Name....1373
OrderByDirection....1373
OrderByNumber....1374
QueryRecord....1374
RecordAlias....1374
ShortName....1375
TranslateEffDtLogic....1375
TranslateExpression....1375
TranslateField....1375
TranslateOption....1376
Type....1376
QueryCriteria Collection....1376
QueryCriteria Collection Methods....1376
First....1377
Item....1377
ItemByName....1377
Next....1378
QueryCriteria Collection Property....1378
Count....1379
QueryCriteria Class....1379
QueryCriteria Class Methods....1379
AddExpr1Expression....1379
AddExpr1Field....1380
AddExpr2Expression....1380
AddExpr2Field1....1381
AddExpr2Field2....1381
AddExpr2List....1382
AddExpr2Subquery....1382
QueryCriteria Class Properties....1383
Expr1Expression....1383
Expr1Field....1383
Expr1Type....1383

Expr2Constant1....1384
Expr2Constant2....1385
Expr2Expression1....1385
Expr2Expression2....1385
Expr2Field1....1385
Expr2Field2....1386
Expr2List....1386
Expr2Subquery....1386
Expr2Type....1386
Logical....1389
LParenLvl....1390
Name....1390
Negation....1390
Operator....1390
R1ExprNum....1392
R2ExprNum....1392
R1ExprType....1392
R2ExprType....1393
RParenLvl....1393
QueryExpression Collection....1393
QueryExpression Collection Methods....1393
First....1393
Item....1394
ItemByName....1394
Next....1395
QueryExpression Collection Property....1395
Count....1395
QueryExpression Class....1395
QueryExpression Class Properties....1396
Aggregate....1396
BindFlag....1396
Decimal....1396
ExpNum....1397
IsXlatExpression....1397
Length....1397
Name....1397
OutputField....1397
RightExprFlag....1397
SelectedField....1398
Text....1398

Type....1398
QueryList Class....1399
QueryList Class Methods....1399
AddListValue....1399
First....1400
Item....1400
Next....1400
QueryList Class Property....1401
Count....1401
QueryListValue Class....1401
QueryListValue Class Properties....1401
IsPrompt....1402
Value....1402
QueryRecordHierarchy Collection....1402
QueryRecordHierarchy Collection Methods....1402
First....1402
Item....1403
ItemByName....1403
Next....1404
QueryRecordHierarchy Collection Property....1404
Count....1404
QueryRecordHierarchy Class....1404
QueryRecordHierarchy Class Properties....1405
Description....1405
Level....1405
Name....1405
ParentFlag....1405
Query Metadata Collection....1406
Query Metadata Collection Methods....1406
First....1406
Item....1406
ItemByName....1407
Next....1407
Query Metadata Collection Property....1408
Count....1408
Query Metadata Class....1408
Query Metadata Class Properties....1408
Name....1409
Value....1409
QueryStatistics Class....1409

QueryStatistics Class Properties 1410
AvgExecTime 1410
AvgFetchTime 1410
AvgNumRows 1410
ExecCount 1410
LastExecDtTm 1410
QuerySecurityProfile Class 1410
QuerySecurityProfile Class Properties 1411
AllowAnyJoin 1411
AllowDistinct 1411
AllowExpressions 1411
AllowSubqueries 1411
AllowUnions 1411
ApprovePrivateQuery 1412
ApprovePublicQuery 1412
CanCreatePublic 1412
CanCreateWorkFlow 1412
CanModifyQuery 1412
CanRunQuery 1413
CanRunToCrystal 1413
CanRunToExcel 1413
LimitUnapproved 1413
MaxInTreeCriteria 1413
MaxJoins 1414
MaxRowsToFetch 1414
MaxUnapprovedRows 1414
QueryDBRecord Collection 1414
QueryDBRecord Collection Methods 1414
First 1414
Item 1415
ItemByName 1415
Next 1416
QueryDBRecord Collection Property 1416
Count 1416
QueryDBRecord Class 1417
QueryDBRecord Class Methods 1417
QueryDBRecordFieldByIndex 1417
QueryDBRecordFieldByName 1417
QueryDBRecord Class Properties 1418
Description 1418

Name....1418
QueryDBRecordFields....1418
RecordHierachy....1419
QueryDBRecordField Collection....1419
QueryDBRecordField Collection Methods....1419
First....1419
Item....1420
ItemByName....1420
Next....1421
Sort....1421
QueryDBRecordField Collection Property....1422
Count....1422
QueryDBRecordField Class....1422
QueryDBRecordField Class Properties....1422
Decimal....1422
Description....1423
Flag....1423
Format....1424
Length....1424
LongName....1425
LookupTableName....1425
LookupTableRecord....1425
Name....1425
Shortname....1425
Type....1425
QueryPrompt Collection....1426
QueryPrompt Collection Methods....1426
First....1426
Item....1426
ItemByName....1427
Next....1427
QueryPrompt Collection Property....1428
Count....1428
QueryPrompt Class....1428
QueryPrompt Properties....1429
EditType....1429
FieldDecimal....1429
FieldFormat....1429
FieldLength....1430
FieldName....1430

FieldType....1430
HeadingText....1431
HeadingType....1431
LangCount....1431
Name....1431
PromptRecordFieldName....1431
PromptTable....1432
UniquePromptName....1432
UseCount....1432
Query Classes Examples....1432
Creating a New Query....1432
Adding Criteria....1434

Chapter 25
Record Class....1441
Understanding Record Class....1441
Shortcut Considerations....1441
Record Methods Used to Create SQL Statements....1442
Data Type of a Record Object....1443
Scope of a Record Object....1443
Record Class Built-in Functions....1443
Record Class Methods....1443
CompareFields....1443
CopyChangedFieldsTo....1444
CopyFieldsTo....1445
DBPatternMatch....1446
Delete....1447
ExecuteEdits....1448
GetField....1450
Insert....1452
SearchClear....1453
SelectByKey....1454
SelectByKeyEffDt....1455
SetDefault....1456
SetEditTable....1457
Update....1459
Record Class Properties....1460
FieldCount....1460
fieldname....1461

IsChanged....1461
IsDeleted....1462
IsEditError....1462
Name....1462
ParentRow....1463
RelLangRecName....1463

Chapter 26
Row Class 1465
Understanding Row Class....1465
Shortcut Considerations 1466
Data Type of a Row Object 1466
Scope of a Row Object 1466
Row Class Built-in Function....1466
Row Class Methods 1466
CopyTo....1467
GetNextEffRow....1467
GetPriorEffRow....1468
GetRecord....1469
GetRowset....1470
scrollname....1471
Row Class Properties 1471
ChildCount....1471
DeleteEnabled....1472
IsChanged....1473
IsDeleted....1473
IsEditError....1474
IsNew....1474
ParentRowset....1475
recname....1475
RecordCount....1475
RowNumber....1476
Selected....1476
Style....1476
Visible....1477

Chapter 27

Rowset Class 1479
Understanding Rowset Class 1479
Shortcut Considerations 1480
Data Type of a Rowset Object 1480
Scope of a Rowset Object 1480
Rowset Class Built-In Functions 1480
Rowset Class Methods 1481
ClearDeletesChanges 1481
CopyTo 1482
DeleteRow 1484
Fill 1485
FillAppend 1488
Flush 1489
FlushRow 1490
GetCurrEffRow 1491
GetNewEffRow 1491
GetRow 1492
HideAllRows 1493
InsertRow 1494
Refresh 1495
Select 1495
SelectNew 1498
SetDefault 1499
ShowAllRows 1500
Sort 1500
Rowset Class Properties 1502
ActiveRowCount 1502
ChangeOnInit 1502
DataAreaCollapsed 1503
DBRecordName 1503
DeleteEnabled 1504
EffDt 1504
EffSeq 1505
InsertEnabled 1505
IsEditError 1506
Level 1507
Name 1507
ParentRow 1507
ParentRowset 1508

RowCount....1508
TopRowNumber....1508

Chapter 28
Search Classes....1509
Understanding Search Class....1509
Using the SearchResults Collection....1510
Understanding SearchResults....1510
Search Classes Hierarchy....1512
Error Handling....1512
Data Type of Search Objects....1514
Scope of the Search Objects....1514
Session Class Methods....1514
GetSearchIndexes....1514
GetSearchQuery....1515
PortalRegistry Class Methods....1515
BuildSearchIndex....1516
GetSearchQuery....1516
SearchQuery Class....1517
SearchQuery Class Methods....1517
Execute....1517
Parse....1518
SearchQuery Class Properties....1519
HitCount....1519
IndexName....1519
KnowledgeBase....1519
Language....1519
ProcessedCount....1520
QueryText....1520
ParseResult Collection....1520
ParseResult Collection Methods....1520
First....1520
Next....1520
ParseResult Collection Properties....1521
ErrorCount....1521
WarningCount....1521
ParseResult Class....1522
ParseResult Class Properties....1522
Message....1522

Severity....1522
SearchResult Collection....1522
SearchResult Collection Methods....1522
First....1523
Item....1523
Next....1523
SearchResult Collection Property....1524
Count....1524
SearchResult Class....1524
SearchResult Class Properties....1524
Key....1524
Score....1525
SearchFields....1525
SearchField Collection....1525
SearchField Collection Methods....1525
First....1525
ItemByName....1525
Next....1526
SearchField Collection Property....1526
Count....1526
SearchField Class....1527
SearchField Class Properties....1527
Name....1527
Value....1527
SearchIndexes Collection....1527
SearchIndexes Collection Methods....1527
First....1528
ItemByName....1528
Next....1528
SearchIndex Collection Property....1529
Count....1529
SearchIndex Class....1529
SearchIndex Class Method....1529
Save....1529
SearchIndex Class Properties....1530
ExtraOptions....1530
FSOpts....1530
HTTPOpts....1530
Languages....1530
Location....1531

Name....1531
RecOpts....1531
Schedules....1531
Type....1531
SearchRecord Options Class....1532
SearchRecord Options Class Properties....1532
Fields....1532
Filter....1533
IncrementalView....1533
RecName....1533
VeggieKey....1533
ZoneOptions....1534
SearchRecordField Collection....1535
SearchRecordField Collection Methods....1535
DeleteItem....1535
First....1536
InsertItem....1536
ItemByName....1537
Next....1537
SearchRecordField Collection Property....1537
Count....1537
SearchRecordField Class....1538
SearchRecordField Class Properties....1538
FieldName....1538
IsAttachment....1538
IsVerityField....1538
IsWordIndex....1539
RecordName....1539
Search Language Collection....1539
Search Language Collection Methods....1539
DeleteItem....1539
First....1540
InsertItem....1540
ItemByName....1540
Next....1541
Search Language Collection Property....1541
Count....1541
Search Language Class....1541
Search Language Class Properties....1542
LanguageCd....1542

MapLanguageCd....1542
Search Schedule Collection....1542
Search Schedule Collection Methods....1542
DeleteItem....1542
First....1543
InsertItem....1543
ItemByName....1543
Next....1544
Search Schedule Collection Property....1544
Count....1544
Search Schedule Class....1545
Search Schedule Class Properties....1545
BuildType....1545
RunCntlId....1545
RunRecurrence....1545
ServerName....1546
Search HTTP Options Class....1546
Search HTTP Options Class Properties....1546
AllowHTTPS....1546
DomainLimit....1547
GlobList....1547
GlobListType....1547
LinkDepth....1547
MIMEList....1548
MIMEListType....1548
ProxyHost....1548
ProxyPort....1548
StartOpts....1549
Search FS Options Class....1549
Search FS Options Class Properties....1549
GlobList....1549
GlobListType....1550
MIMEList....1550
MIMEListType....1550
StartOpts....1550
Search Start Options Collection....1551
Search Start Options Collection Methods....1551
DeleteItem....1551
First....1552
InsertItem....1552

ItemByName....1552
Next....1553
Search Start Options Collection Property....1553
Count....1553
Search Start Options Class....1553
Search Start Options Class Properties....1554
IsDomainRestricted....1554
IsHostRestricted....1554
Value....1554
Search API Examples....1554
General Purpose Search API Example....1554
Portal Search API Example....1556

Chapter 29
Session Class....1559
Understanding Session Class....1559
Security and Access to the PeopleSoft System....1560
Error Handling....1560
Displaying Error Messages....1561
Regional Settings....1562
Trace Settings....1562
Session Object Declaration....1563
State Considerations....1563
Considerations for Declaring Collections....1563
Scope of a Session Object....1564
Session Class Built-in Function....1564
Session Class Methods....1564
Connect....1564
Disconnect....1566
API Instantiation Methods....1566
Session Class Properties....1567
ErrorPending....1567
PSMessages....1568
PSMessagesMode....1568
RegionalSettings....1569
Repository....1569
SuspendFormatting....1569
TraceSettings....1569
WarningPending....1570

Accessing a PSMessages Collection 1570
PSMessages Collection Methods 1570
DeleteAll 1571
DeleteItem 1571
First 1571
Item 1572
Next 1572
PSMessages Collection Property 1572
Count 1573
PSMessage Class Properties 1573
ExplainText 1573
MessageNumber 1573
MessageSetNumber 1573
MessageType 1574
Source 1574
Text 1576
Regional Settings Class Properties 1577
ClientTimeZone 1577
CurrencyFormat 1577
CurrencySymbol 1578
DateFormat 1578
DateSeparator 1578
DecimalSymbol 1578
DigitGroupingSymbol 1579
LanguageCD 1579
UseLocalTime 1580
1159Separator 1580
2359Separator 1580
Trace Setting Class Properties 1580
API 1580
COBOLStmtTimings 1581
ConnDisRollbackCommit 1581
DBSpecificCalls 1581
ManagerInfo 1581
NetworkServices 1581
NonSSBs 1581
OutputUNICODE 1582
PCExtFcnCalls 1582
PCFcnReturnValues 1582
PCFetchedValues 1582

PCIntFcnCalls....1582
PCListProgram....1583
PCParameterValues....1583
PCProgramStatements....1583
PCStack....1583
PCStartOfPrograms....1583
PCTraceProgram....1583
PCVariableAssignments....1584
RowFetch....1584
SQLStatement....1584
SQLStatementVariables....1584
SSBs....1584
SybBindInfo....1585
SybFetchInfo....1585
TraceFile....1585

Chapter 30
SOAPDoc Class 1587
Understanding SOAPDoc Class 1587
SOAP Message Exchange Model 1588
SOAP Message Document 1588
The SOAPDoc Class 1590
SOAPDoc Object Creation 1590
SOAP Header Section 1590
SOAP Envelope Section 1591
SOAP Body Section 1591
SOAP Method Section 1591
SOAPDoc Section Access 1592
PeopleCode to Create a SOAPDoc 1592
SOAPDoc Fault Section 1593
The ValidateSOAPDoc Method 1593
SOAPDoc Objects and Messages 1594
Data Type of a SOAPDoc Object 1594
Scope of a SOAPDoc Object 1594
SOAPDoc Built-In Functions....1594
SOAPDoc Class Methods 1595
AddBody....1595
AddEnvelope....1596
AddFault....1597

AddHeader....1598
AddMethod....1600
AddParm....1601
GetHeaderNode....1602
GetParmName....1603
GetParmValue....1603
ValidateSOAPDoc....1604
SOAPDoc Class Properties....1605
BodyNode....1605
EnvelopeNode....1606
FaultCode....1606
FaultString....1606
MethodName....1606
MethodNode....1607
ParmCount....1607
XmlDoc....1607
SOAPDoc Class Examples....1608
Creating a SOAP Document....1608
Reading an Existing SOAP Document....1608
Using SOAP XML Text....1609

Chapter 31
SQL Class....1613
Understanding SQL Class....1613
Using Record Class SQL....1615
Creating a SQL Definition....1615
Binding and Executing of SQL Statements....1615
Fetching From a Select....1616
Using Array of Any for Bind Values or Fetch Results....1617
Understanding Different "Styles" of SELECT....1617
Reusing a Cursor....1618
Using the ReuseCursor Property....1618
Reusing a Cursor Without the ReuseCursor Property....1619
Understanding SQL Objects and PeopleSoft Application Engine Programs....1621
Declaring a SQL Object....1621
Scope of an SQL Object....1621
SQL Class Built-in Functions....1621
SQL Class Methods....1622
Close....1622

Execute....1622
Fetch....1624
Open....1626
SQL Class Properties....1627
BulkMode....1627
IsOpen....1628
LTrim....1629
ReuseCursor....1629
RowsAffected....1630
Status....1630
TraceName....1631
Value....1632

Chapter 32
SyncServer Class....1633
Understanding Synchronization Server Events and the SyncServer Classes....1633
Understanding the Synchronization Events....1633
Understanding the SyncServer Class....1635
Considerations Using Synchronization Events and the SyncServer Class....1635
Using the Synchronization Server Events....1636
Using OnConflict....1636
Using OnSelect....1637
Using OnValidate....1639
Using Attachment Events....1640
Scope of SyncServer Objects....1642
SyncServer Class....1642
SyncServer Class Methods....1643
AddReference....1643
AddReferenceType....1643
SyncServer Class Properties....1644
ClientPlatform....1644
ConflictAlgorithm....1644
ConflictStatus....1645
ExcludeProperty....1646
ExportLocation....1646
PropertyValue....1646
SessionID....1647
Synchronizing....1647
SyncIDs....1648

TypeID....1648
validateID....1648
validateRowCount....1649
validateVersion....1649
Advanced Considerations for Synchronization....1649
Using OnValidateSet....1649
Using OnGetDefinition....1650

Chapter 33
Tree Classes....1653
Understanding Tree Classes....1653
Relationships Between Different Tree Classes....1654
Collections in the Tree Classes....1656
Error Handling With Trees....1658
Leaves and Nodes Insert Verification....1659
Data Types for Tree Objects....1659
Scope of the Tree Objects....1659
Tree Classes Implementation....1660
Session Class Methods....1663
FindTree....1663
FindTreeStructure....1664
GetTree....1665
GetTreeStructure....1666
Branch Collection....1666
Branch Collection Method....1666
Item....1666
Branch Collection Properties....1667
Count....1667
First....1667
Last....1668
Next....1668
Leaf Class....1668
Leaf Class Methods....1668
Cut....1668
Delete....1669
DeleteByRange....1669
InsertDynSib....1670
InsertSib....1670
MoveAsChild....1671

MoveAsChildByName....1672
MoveAsSib....1672
MoveAsSibByRange....1673
PasteSib....1673
Leaf Class Properties....1674
DisplayLevelNumber....1674
Dynamic....1674
HasNextSib....1674
HasPrevSib....1675
IsChanged....1675
IsCut....1675
IsDeleted....1675
IsInserted....1675
NextSib....1676
Parent....1676
PrevSib....1676
RangeFrom....1677
RangeTo....1677
TreeBranchName....1677
TreeEffDt....1677
TreeName....1677
TreeSetId....1677
TreeUserKeyValue....1678
Level Collection....1678
Level Collection Methods....1678
Add....1678
Item....1678
Remove....1679
Save....1679
Level Collection Properties....1679
Count....1680
First....1680
Last....1680
Next....1680
Level Class....1680
Level Class Methods....1680
Create....1681
Delete....1681
Level Class Properties....1681
AllValuesAudit....1682

Description....1682
Name....1682
Number....1682
TreeBranchName....1682
TreeEffDt....1682
TreeName....1683
TreeSetId....1683
TreeUserKeyValue....1683
Node Class....1683
Node Class Methods....1683
Branch....1684
Cut....1684
Delete....1684
DeleteByName....1685
Expand....1685
InsertChildLeaf....1685
InsertChildNode....1686
InsertChildRecord....1687
InsertDynChildLeaf....1688
InsertSib....1688
InsertSibRecord....1689
MoveAsChild....1690
MoveAsChildByName....1690
MoveAsSib....1691
MoveAsSibByName....1691
PasteChild....1692
PasteSib....1692
RefreshDescription....1693
Rename....1693
SwitchLevel....1694
Unbranch....1694
Node Class Properties....1695
AllChildCount....1695
AllChildNodeCount....1695
ChildLeafCount....1695
ChildNodeCount....1695
Description....1696
DisplayLevelNumber....1696
FirstChildLeaf....1696
FirstChildNode....1696

HasChildLeaves....1696
HasChildNodes....1697
HasChildren....1697
HasNextSib....1697
HasPrevSib....1697
IsBranched....1697
IsChanged....1697
IsCut....1698
IsDeleted....1698
IsInserted....1698
IsRoot....1698
LastChildLeaf....1698
LastChildNode....1699
LevelNumber....1699
Name....1699
NextSib....1699
Parent....1700
PrevSib....1700
State....1700
TreeBranchName....1701
TreeEffDt....1701
TreeName....1701
TreeSetId....1701
TreeUserKeyValue....1701
Type....1702
Tree Class....1702
Tree Class Methods....1703
Audit....1703
AuditByName....1703
Close....1704
Copy....1705
Create....1707
Delete....1708
Exists....1709
FindLeaf....1710
FindNextDisplayObject....1710
FindNode....1711
FindPrevDisplayObject....1712
FindRoot....1713
InsertRoot....1714

LeafExists....1714
Open....1715
OpenWholeTree....1717
NodeExists....1718
Rename....1718
Save....1719
SaveAs....1720
SaveAsDraft....1721
SaveDraft....1722
Tree Class Properties....1723
AllValues....1723
AuditDetails....1723
Branches....1724
BranchLevel....1724
BranchName....1724
Category....1724
CheckLeafUserData....1725
Description....1725
DisplayLeaf....1725
DisplayNode....1726
DuplicateLeaves....1726
EffDt....1726
HasDetailRanges....1726
IsBranched....1727
IsChanged....1727
IsOpen....1727
IsQueryTree....1728
IsReadOnly....1728
IsValid....1728
IsWholeTree....1728
KeyBranchName....1729
KeyEffDt....1729
KeyName....1729
KeySetId....1729
KeyUserKeyValue....1730
LeafCount....1730
LeafOnClipboard....1730
LevelCount....1730
Levels....1731
LevelUse....1731

Name....1731
NodeCount....1732
NodeOnClipboard....1732
ParentLevel....1732
ParentName....1732
PerformanceMethod....1733
PerformanceSelector....1733
PerformanceSelectorOption....1733
SetID....1734
Status....1734
Structure....1734
StructureName....1735
TotalNodeCount....1735
UserKeyValue....1735
Tree Collection....1736
Tree Collection Method....1737
Item....1737
Tree Collection Properties....1738
Count....1738
First....1739
Last....1739
Next....1739
Tree Structure Class....1739
Tree Structure Class Methods....1739
Close....1739
Copy....1740
Create....1740
Delete....1741
Open....1741
Rename....1741
Save....1742
Tree Structure Class Properties....1742
Description....1742
DetailComponent....1742
DetailField....1743
DetailMenu....1743
DetailMenuBar....1743
DetailMenuItem....1744
DetailMultiNavigate....1744
DetailPage....1744

DetailRecord....1745
IndirectionMethod....1745
KeyName....1747
LevelComponent....1747
LevelMenu....1748
LevelMenuBar....1748
LevelMenuItem....1748
LevelPage....1748
LevelRecord....1749
Name....1749
NodeComponent....1749
NodeField....1750
NodeMenu....1750
NodeMenuBar....1750
NodeMenuItem....1750
NodeMultiNavigate....1751
NodePage....1751
NodeRecord....1751
NodeUserKeyField....1752
SummarySetId....1752
SummaryLevelNumber....1752
SummaryTreeName....1752
SummaryUserKeyValue....1753
Type....1753
Tree Structure Collection....1753
Tree Structure Collection Method....1754
Item....1754
Tree Structure Collection Properties....1755
Count....1755
First....1755
Last....1755
Next....1755
Traverse Tree Hierarchy Example....1755

Chapter 34
XmlDoc Class....1757
Understanding XmlDoc Class....1757
When to Use an XmlDoc Object....1758
XmlDoc Object Creation....1758

XmlNode Class Considerations 1759
Accessing and Traversing an XmlNode Object 1760
Error Handling 1760
SOAPDoc Object Considerations 1760
Scope of XmlDoc and XmlNode Objects 1761
Data Type of an XmlDoc or XmlNode Object 1761
XmlDoc Class Built-in Functions 1761
XmlDoc Methods 1762
CopyRowset 1762
CopyToPSFTMessage 1763
CopyToRowset 1765
CreateDocumentElement 1767
CreateDocumentType 1768
GenFormattedXmlString 1769
GenXmlString 1770
GetElementsByTagName 1770
LoadIBContent 1771
ParseXmlFromURL 1773
ParseXmlString 1774
XmlDoc Properties 1775
DocumentElement 1775
IsNull 1776
XmlNode 1776
XmlNode Methods 1776
AddAttribute 1776
AddAttributeNS 1777
AddCDataSection 1778
AddComment 1779
AddElement 1780
AddElementNS 1781
AddEntityReference 1782
AddNode 1783
AddProcessInstruction 1784
AddText 1785
CopyNode 1786
FindNode 1787
FindNodes 1788
GenXmlString 1789
GetAttributeName 1789
GetAttributeValue 1790

GetCDataValue....1790
GetCDataValues....1791
GetChildNode....1792
GetElement....1793
GetElements....1793
GetElementsByTagName....1794
GetElementsByTagNameNS....1795
InsertCDataSection....1796
InsertComment....1797
InsertElement....1799
InsertElementNS....1800
InsertEntityReference....1801
InsertNode....1802
InsertProcessInstruction....1804
InsertText....1805
RemoveAllChildNode....1806
RemoveChildNode....1807
XmlNode Properties....1808
AttributesCount....1809
ChildNodeCount....1809
Index....1809
IsNull....1809
LocalName....1810
NamespaceURI....1810
NextSibling....1810
NodeName....1810
NodePath....1811
NodeType....1811
NodeValue....1811
ParentNode....1811
Prefix....1812
PreviousSibling....1812

Appendix A

Appendix A: Cheat Sheets....1813

Class List Cheat Sheet....1813
Summary of PeopleCode Classes....1813
Session Class Methods and Properties....1859
Component Interface Class Methods and Properties....1862

Search Classes Methods and Properties 1866
PortalRegistry Classes Methods and Properties 1873
Query Classes Methods and Properties 1893
Tree Classes Methods and Properties 1910
Mobile Classes Methods and Properties 1920
Internet Script Classes Methods and Properties 1924
Mapping of Functions to Methods and Properties 1928
Mapping of Old Names to New Names 1931
Deprecated Functions 1933
Deprecated Methods and Properties. 1934
PeopleCode Syntax Cheat Sheet 1934

Appendix B
Appendix B: Viewing Trees From Application Pages 1939
Understanding View Trees 1939
Invoking View Trees From Application Pages 1942
View Trees From Application Example−Without Multi-Node Selection (Method A) 1945
View Trees From Application Example−With Multi-Node Selection (Method B) 1946

Glossary of PeopleSoft Terms....1949

Index 1961

About This PeopleBook

PeopleBooks provide you with the information that you need to implement and use PeopleSoft applications.

This preface discusses:

- PeopleSoft application prerequisites.
- PeopleSoft application fundamentals.
- Related documentation.
- Typographical elements and visual cues.
- Comments and suggestions.
- Common elements in PeopleBooks.

Note. PeopleBooks document only page elements that require additional explanation. If a page element is not documented with the process or task in which it is used, then either it requires no additional explanation or it is documented with common elements for the section, chapter, PeopleBook, or product line. Elements that are common to all PeopleSoft applications are defined in this preface.

PeopleSoft Application Prerequisites

To benefit fully from the information that is covered in these books, you should have a basic understanding of how to use PeopleSoft applications.

See *Using PeopleSoft Applications.*

You might also want to complete at least one PeopleSoft introductory training course.

You should be familiar with navigating the system and adding, updating, and deleting information by using PeopleSoft windows, menus, and pages. You should also be comfortable using the World Wide Web and the Microsoft Windows or Windows NT graphical user interface.

These books do not review navigation and other basics. They present the information that you need to use the system and implement your PeopleSoft applications most effectively.

PeopleSoft Application Fundamentals

Each application PeopleBook provides implementation and processing information for your PeopleSoft database. However, additional, essential information describing the setup and design of your system appears in a companion volume of documentation called the application fundamentals PeopleBook. Each PeopleSoft product line has its own version of this documentation.

The application fundamentals PeopleBook consists of important topics that apply to many or all PeopleSoft applications across a product line. Whether you are implementing a single application, some combination of applications within the product line, or the entire product line, you should be familiar with the contents of this central PeopleBook. It is the starting point for fundamentals, such as setting up control tables and administering security.

Related Documentation

This section discusses how to:

- Obtain documentation updates.
- Order printed documentation.

Obtaining Documentation Updates

You can find updates and additional documentation for this release, as well as previous releases, on the PeopleSoft Customer Connection Website. Through the Documentation section of PeopleSoft Customer Connection, you can download files to add to your PeopleBook Library. You'll find a variety of useful and timely materials, including updates to the full PeopleSoft documentation that is delivered on your PeopleBooks CD-ROM.

Important! Before you upgrade, you must check PeopleSoft Customer Connection for updates to the upgrade instructions. PeopleSoft continually posts updates as the upgrade process is refined.

See Also

PeopleSoft Customer Connection Website, http://www.peoplesoft.com/corp/en/login.asp

Ordering Printed Documentation

You can order printed, bound volumes of the complete PeopleSoft documentation that is delivered on your PeopleBooks CD-ROM. PeopleSoft makes printed documentation available for each major release shortly after the software is shipped. Customers and partners can order printed PeopleSoft documentation by using any of these methods:

- Web
- Telephone
- Email

Web

From the Documentation section of the PeopleSoft Customer Connection Website, access the PeopleSoft Press Website under the Ordering PeopleBooks topic. The PeopleSoft Press Website is a joint venture between PeopleSoft and Consolidated Publications Incorporated (CPI), the book print vendor. Use a credit card, money order, cashier's check, or purchase order to place your order.

Telephone

Contact CPI at 800 888 3559.

Email

Send email to CPI at psoftpress@cc.larwood.com.

See Also

PeopleSoft Customer Connection Website, http://www.peoplesoft.com/corp/en/login.asp

Typographical Conventions and Visual Cues

This section discusses:

- Typographical conventions.
- Visual cues.

Typographical Conventions

The following table contains the typographical conventions that are used in PeopleBooks:

Typographical Convention or Visual Cue	Description
Bold	Indicates PeopleCode function names, method names, language constructs, and PeopleCode reserved words that must be included literally in the function call.
Italics	Indicates field values, emphasis, and PeopleSoft or other book-length publication titles. In PeopleCode syntax, italic items are placeholders for arguments that your program must supply. We also use italics when we refer to words as words or letters as letters, as in the following: Enter the number 0, not the letter *O*.
KEY+KEY	Indicates a key combination action. For example, a plus sign (+) between keys means that you must hold down the first key while you press the second key. For ALT+W, hold down the ALT key while you press W.
`Monospace font`	Indicates a PeopleCode program or other code example.
(quotation marks)	Indicate chapter titles in cross-references and words that are used differently from their intended meanings.

Typographical Convention or Visual Cue	Description
. . . (ellipses)	Indicate that the preceding item or series can be repeated any number of times in PeopleCode syntax.
{ } (curly braces)	Indicate a choice between two options in PeopleCode syntax. Options are separated by a pipe (|).
[ ] (square brackets)	Indicate optional items in PeopleCode syntax.
& (ampersand)	When placed before a parameter in PeopleCode syntax, an ampersand indicates that the parameter is an already instantiated object. Ampersands also precede all PeopleCode variables.
(ISO)	Information that applies to a specific country, to the U.S. federal government, or to the education and government market, is preceded by a three-letter code in parentheses. The code for the U.S. federal government is USF; the code for education and government is E&G, and the country codes from the International Standards Organization are used for specific countries. Here is an example: (GER) If you're administering German employees, German law requires you to indicate special nationality and citizenship information for German workers using nationality codes established by the German DEUEV Directive.
Cross-references	PeopleBooks provide cross-references either below the heading See Also or on a separate line preceded by the word *See*. Cross-references lead to other documentation that is pertinent to the immediately preceding documentation.

Visual Cues

PeopleBooks contain the following visual cues.

Notes

Notes indicate information that you should pay particular attention to as you work with the PeopleSoft system.

Note. Example of a note.

A note that is preceded by *Important!* is crucial and includes information that concerns what you must do for the system to function properly.

Important! Example of an important note.

Warnings

Warnings indicate crucial configuration considerations. Pay close attention to warning messages.

Warning! Example of a warning.

Comments and Suggestions

Your comments are important to us. We encourage you to tell us what you like, or what you would like to see changed about PeopleBooks and other PeopleSoft reference and training materials. Please send your suggestions to:

PeopleSoft Product Documentation Manager PeopleSoft, Inc. 4460 Hacienda Drive Pleasanton, CA 94588

Or send email comments to doc@peoplesoft.com.

While we cannot guarantee to answer every email message, we will pay careful attention to your comments and suggestions.

Common Elements in These PeopleBooks

As of Date	The last date for which a report or process includes data.
Business Unit	An ID that represents a high-level organization of business information. You can use a business unit to define regional or departmental units within a larger organization.
Description	Enter up to 30 characters of text.
Effective Date	The date on which a table row becomes effective; the date that an action begins. For example, to close out a ledger on June 30, the effective date for the ledger closing would be July 1. This date also determines when you can view and change the information. Pages or panels and batch processes that use the information use the current row.
Once, Always, and **Don't Run**	Select Once to run the request the next time the batch process runs. After the batch process runs, the process frequency is automatically set to Don't Run. Select Always to run the request every time the batch process runs. Select Don't Run to ignore the request when the batch process runs.
Report Manager	Click to access the Report List page, where you can view report content, check the status of a report, and see content detail messages (which show you a description of the report and the distribution list).

Process Monitor	Click to access the Process List page, where you can view the status of submitted process requests.
Run	Click to access the Process Scheduler request page, where you can specify the location where a process or job runs and the process output format.
Request ID	An ID that represents a set of selection criteria for a report or process.
User ID	An ID that represents the person who generates a transaction.
SetID	An ID that represents a set of control table information, or TableSets. TableSets enable you to share control table information and processing options among business units. The goal is to minimize redundant data and system maintenance tasks. When you assign a setID to a record group in a business unit, you indicate that all of the tables in the record group are shared between that business unit and any other business unit that also assigns that setID to that record group. For example, you can define a group of common job codes that are shared between several business units. Each business unit that shares the job codes is assigned the same setID for that record group.
Short Description	Enter up to 15 characters of text.

See Also

Using PeopleSoft Applications

PeopleSoft Process Scheduler

PeopleCode Reference Preface

This preface introduces the PeopleCode Reference PeopleBook.

PeopleCode Reference

This PeopleBook is a complete reference to PeopleCode, the proprietary language used in the development of PeopleSoft applications. Its chapters describe the syntax and fundamental elements of the PeopleCode language.

As a licensee of PeopleTools, you are licensed to use the base portal technology, which is limited to navigation to licensed PeopleSoft applications. If you want to register additional non-PeopleSoft content, customize your homepage, or create any pagelets, you must license PeopleSoft Enterprise Portal.

The “About These PeopleBooks” preface contains general product line information, such as related documentation, common page elements, and typographical conventions. This preface also contains a glossary with useful terms that are used in PeopleBooks.

See also PeopleCode Developer’s Guide

CHAPTER 1

PeopleCode Built-in Functions

This chapter provides a reference to PeopleCode built-in functions and language constructs and discusses:

- PeopleCode Typographical Conventions
- Functions by category
- PeopleCode built-in functions and language constructs

PeopleCode Typographical Conventions

Throughout this book, we use typographical conventions to distinguish between different elements of the PeopleCode language, such as bold to indicate function names, italics for arguments, and so on.

Please take a moment to review the following typographical cues:

Font Type	Description
`monospace font`	Indicates a PeopleCode program or other example
Keyword	In PeopleCode syntax, items in **keyword** font indicate function names, method names, language constructs, and PeopleCode reserved words that must be included literally in the function call.
Variable	In PeopleCode syntax, items in *variable* font are placeholders for arguments that your program must supply.
...	In PeopleCode syntax, ellipses indicate that the preceding item or series can be repeated any number of times.
{*Option1*|*Option2*}	In PeopleCode syntax, when there is a choice between two options, the options are enclosed in curly braces and separated by a pipe.

Font Type	Description
[ ]	In PeopleCode syntax optional items are enclosed in square brackets.
&Parameter	In PeopleCode syntax an ampersand before a parameter indicates that the parameter is an already instantiated object.

Functions by Category

The following topics subdivide the PeopleCode built-in functions by functional category and provide links from within each category to the reference entries.

APIs

Chapter 1, PeopleCode Built-in Functions, GetSession , page 269

Chapter 3, System Variables , %Session , page 577

Application Classes

Chapter 1, PeopleCode Built-in Functions, CollectGarbage , page 70

Application Engine

Chapter 1, PeopleCode Built-in Functions, CallAppEngine , page 58

Chapter 1, PeopleCode Built-in Functions, CommitWork , page 71

Chapter 1, PeopleCode Built-in Functions, GetAESection , page 233

Arrays

Chapter 1, PeopleCode Built-in Functions, CreateArray , page 92

Chapter 1, PeopleCode Built-in Functions, CreateArrayAny , page 93

Chapter 1, PeopleCode Built-in Functions, CreateArrayRept , page 94

Chapter 1, PeopleCode Built-in Functions, Split , page 441

Attachment

Chapter 1, PeopleCode Built-in Functions, AddAttachment , page 31

Chapter 1, PeopleCode Built-in Functions, CopyAttachments , page 88

Chapter 1, PeopleCode Built-in Functions, DeleteAttachment , page 135

Chapter 1, PeopleCode Built-in Functions, GetAttachment , page 234

Chapter 1, PeopleCode Built-in Functions, PutAttachment , page 353

Chapter 1, PeopleCode Built-in Functions, ViewAttachment , page 494

See Also
Files page 10, Images page 11.

Bulk

Chapter 1, PeopleCode Built-in Functions, BulkDeleteField , page 49

Chapter 1, PeopleCode Built-in Functions, BulkInsertField , page 51

Chapter 1, PeopleCode Built-in Functions, BulkModifyPageFieldOrder , page 54

Chapter 1, PeopleCode Built-in Functions, BulkUpdateIndexes , page 56

Business Interlink

Chapter 1, PeopleCode Built-in Functions, GetBiDoc , page 236

Chapter 1, PeopleCode Built-in Functions, GetInterlink , page 250

Chapter 3, System Variables , %MaxInterlinkSize , page 570

Chart

Chapter 1, PeopleCode Built-in Functions, CreateObject , page 101

Chapter 1, PeopleCode Built-in Functions, GetChart , page 239

Chapter 1, PeopleCode Built-in Functions, GetChartURL , page 240

ChartField

Chapter 1, PeopleCode Built-in Functions, RenameDBField , page 368

Chapter 1, PeopleCode Built-in Functions, RenamePage , page 370

Chapter 1, PeopleCode Built-in Functions, RenameRecord , page 371

Chapter 1, PeopleCode Built-in Functions, SetDBFieldAuxFlag , page 405

Chapter 1, PeopleCode Built-in Functions, SetDBFieldCharDefn , page 407

Chapter 1, PeopleCode Built-in Functions, SetDBFieldFormat , page 408

Chapter 1, PeopleCode Built-in Functions, SetDBFieldFormatLength , page 410

Chapter 1, PeopleCode Built-in Functions, SetDBFieldLabel , page 411

Chapter 1, PeopleCode Built-in Functions, SetDBFieldLength , page 412

Chapter 1, PeopleCode Built-in Functions, SetDBFieldNotUsed , page 414

Chapter 1, PeopleCode Built-in Functions, SetPageFieldPageFieldName , page 423

Chapter 1, PeopleCode Built-in Functions, SetRecFieldEditTable , page 425

Chapter 1, PeopleCode Built-in Functions, SetRecFieldKey , page 427

Component Buffer

Chapter 1, PeopleCode Built-in Functions, ActiveRowCount , page 29

Chapter 1, PeopleCode Built-in Functions, AddKeyListItem , page 36

Chapter 1, PeopleCode Built-in Functions, CompareLikeFields , page 73

Chapter 1, PeopleCode Built-in Functions, ClearKeyList , page 67

Chapter 1, PeopleCode Built-in Functions, ComponentChanged , page 75

Chapter 1, PeopleCode Built-in Functions, CopyFields , page 89

Chapter 1, PeopleCode Built-in Functions, CopyRow , page 90

Chapter 1, PeopleCode Built-in Functions, CurrentLevelNumber , page 116

Chapter 1, PeopleCode Built-in Functions, CurrentRowNumber , page 116

Chapter 1, PeopleCode Built-in Functions, DeleteRecord , page 139

Chapter 1, PeopleCode Built-in Functions, DeleteRow , page 140

Chapter 1, PeopleCode Built-in Functions, DiscardRow , page 146

Chapter 1, PeopleCode Built-in Functions, ExpandBindVar , page 177

Chapter 1, PeopleCode Built-in Functions, ExpandEnvVar , page 178

Chapter 1, PeopleCode Built-in Functions, ExpandSqlBinds , page 178

Chapter 1, PeopleCode Built-in Functions, FetchValue , page 181

Chapter 1, PeopleCode Built-in Functions, FieldChanged , page 182

Chapter 1, PeopleCode Built-in Functions, GetNextNumber , page 257

Chapter 1, PeopleCode Built-in Functions, GetNextNumberWithGaps , page 259

Chapter 1, PeopleCode Built-in Functions, GetRelField , page 266

Chapter 1, PeopleCode Built-in Functions, GetSetId , page 270

Chapter 1, PeopleCode Built-in Functions, InsertRow , page 297

Chapter 1, PeopleCode Built-in Functions, PriorValue , page 349

Chapter 1, PeopleCode Built-in Functions, RecordChanged , page 358

Chapter 1, PeopleCode Built-in Functions, RecordDeleted , page 359

Chapter 1, PeopleCode Built-in Functions, RecordNew , page 361

Chapter 1, PeopleCode Built-in Functions, RowFlush , page 383

Chapter 1, PeopleCode Built-in Functions, SetChannelStatus , page 401

Chapter 1, PeopleCode Built-in Functions, SetDefault , page 415

Chapter 1, PeopleCode Built-in Functions, SetDefaultAll , page 416

Chapter 1, PeopleCode Built-in Functions, SetTempTableInstance , page 431

Chapter 1, PeopleCode Built-in Functions, StopFetching , page 449

Chapter 1, PeopleCode Built-in Functions, TotalRowCount , page 462

Chapter 1, PeopleCode Built-in Functions, TreeDetailInNode , page 480

Chapter 1, PeopleCode Built-in Functions, UpdateSysVersion , page 490

Chapter 1, PeopleCode Built-in Functions, UpdateValue , page 490

Chapter 3, System Variables , %Component , page 564

Chapter 3, System Variables , %Menu , page 571

Chapter 3, System Variables , %Mode , page 571

Chapter 3, System Variables , %OperatorClass , page 572

Chapter 2, Meta-SQL , %Table , page 551

Chapter 2, Meta-SQL , %TruncateTable , page 557

See Also

Data Buffer Access page 6.

Component Interface

Chapter 1, PeopleCode Built-in Functions, GetMethodNames , page 254

Chapter 1, PeopleCode Built-in Functions, GetSession , page 269

Chapter 1, PeopleCode Built-in Functions, StartWork , page 447

Chapter 3, System Variables , %CompIntfcName , page 565

Conversion

Chapter 1, PeopleCode Built-in Functions, Char , page 64

Chapter 1, PeopleCode Built-in Functions, Code , page 69

Chapter 1, PeopleCode Built-in Functions, Codeb , page 70

Chapter 1, PeopleCode Built-in Functions, ConvertChar , page 79

Chapter 1, PeopleCode Built-in Functions, NumberToString , page 337

Chapter 1, PeopleCode Built-in Functions, String , page 451

Chapter 1, PeopleCode Built-in Functions, Value , page 493

Currency and Financial

Chapter 1, PeopleCode Built-in Functions, Amortize , page 44

Chapter 1, PeopleCode Built-in Functions, ConvertCurrency , page 83

Chapter 1, PeopleCode Built-in Functions, RoundCurrency , page 382

Chapter 1, PeopleCode Built-in Functions, SinglePaymentPV , page 439

Chapter 1, PeopleCode Built-in Functions, UniformSeriesPV , page 489

Chapter 3, System Variables , %Currency , page 566

Current Date and Time

Chapter 2, Meta-SQL , %CurrentDateIn , page 520

Chapter 2, Meta-SQL , %CurrentDateOut , page 520

Chapter 2, Meta-SQL , %CurrentDateTimeIn , page 520

Chapter 2, Meta-SQL , %CurrentDateTimeOut , page 520

Chapter 2, Meta-SQL , %CurrentTimeIn , page 521

Chapter 2, Meta-SQL , %CurrentTimeOut , page 521

See Also

Date and Time page 7, SQL Date and Time page 21.

Custom Display Formats

Chapter 1, PeopleCode Built-in Functions, GetStoredFormat , page 273

Chapter 1, PeopleCode Built-in Functions, SetDisplayFormat , page 419

Database and Platform

Chapter 3, System Variables , %DbName , page 567

Chapter 3, System Variables , %DbType , page 567

Data Buffer Access

Chapter 1, PeopleCode Built-in Functions, CreateRecord , page 105

Chapter 1, PeopleCode Built-in Functions, CreateRowset , page 107

Chapter 1, PeopleCode Built-in Functions, FlushBulkInserts , page 191

Chapter 1, PeopleCode Built-in Functions, GetField , page 242

Chapter 1, PeopleCode Built-in Functions, GetLevel0 , page 253

Chapter 1, PeopleCode Built-in Functions, GetRecord , page 265

Chapter 1, PeopleCode Built-in Functions, GetRow , page 267

Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268

See Also

Component Buffer page 4.

Date and Time

Chapter 1, PeopleCode Built-in Functions, AddToDate , page 39

Chapter 1, PeopleCode Built-in Functions, AddToDateTime , page 40

Chapter 1, PeopleCode Built-in Functions, AddToTime , page 40

Chapter 1, PeopleCode Built-in Functions, ConvertDatetimeToBase , page 85

Chapter 1, PeopleCode Built-in Functions, Date , page 117

Chapter 1, PeopleCode Built-in Functions, Date3 , page 118

Chapter 1, PeopleCode Built-in Functions, DatePart , page 118

Chapter 1, PeopleCode Built-in Functions, DateTime6 , page 119

Chapter 1, PeopleCode Built-in Functions, DateTimeToHTTP , page 120

Chapter 1, PeopleCode Built-in Functions, DateTimeToLocalizedString , page 121

Chapter 1, PeopleCode Built-in Functions, DateTimeToTimeZone , page 124

Chapter 1, PeopleCode Built-in Functions, DateTimeValue , page 125

Chapter 1, PeopleCode Built-in Functions, DateValue , page 126

Chapter 1, PeopleCode Built-in Functions, Day , page 127

Chapter 1, PeopleCode Built-in Functions, Days , page 128

Chapter 1, PeopleCode Built-in Functions, Days360 , page 128

Chapter 1, PeopleCode Built-in Functions, Days365 , page 129

Chapter 1, PeopleCode Built-in Functions, FormatDateTime , page 192

Chapter 1, PeopleCode Built-in Functions, GetCalendarDate , page 237

Chapter 1, PeopleCode Built-in Functions, Hour , page 291

Chapter 1, PeopleCode Built-in Functions, IsDaylightSavings , page 302

Chapter 1, PeopleCode Built-in Functions, Minute , page 326

Chapter 1, PeopleCode Built-in Functions, Month , page 327

Chapter 1, PeopleCode Built-in Functions, Second , page 393

Chapter 1, PeopleCode Built-in Functions, Time , page 458

Chapter 1, PeopleCode Built-in Functions, Time3 , page 458

Chapter 1, PeopleCode Built-in Functions, TimePart , page 459

Chapter 1, PeopleCode Built-in Functions, TimeToTimeZone , page 460

Chapter 1, PeopleCode Built-in Functions, TimeValue , page 461

Chapter 1, PeopleCode Built-in Functions, TimeZoneOffset , page 461

Chapter 1, PeopleCode Built-in Functions, Weekday , page 501

Chapter 1, PeopleCode Built-in Functions, Year , page 509

Chapter 3, System Variables , %AsOfDate , page 563

Chapter 3, System Variables , %ClientDate , page 564

Chapter 3, System Variables , %ClientTimeZone , page 564

Chapter 3, System Variables , %Date , page 566

Chapter 2, Meta-SQL , %DateAdd , page 521

Chapter 2, Meta-SQL , %DateDiff , page 521

Chapter 3, System Variables , %DateTime , page 567

Chapter 2, Meta-SQL , %DateTimeDiff , page 523

Chapter 2, Meta-SQL , %DateTimeIn , page 524

Chapter 2, Meta-SQL , %DateTimeOut , page 525

Chapter 2, Meta-SQL , %DTTM , page 526

Chapter 3, System Variables , %PermissionLists , page 575

Chapter 3, System Variables , %PerfTime, page 574

Chapter 3, System Variables , %ServerTimeZone , page 577

Chapter 3, System Variables , %Time , page 580

Chapter 2, Meta-SQL , %TextIn , page 553

See Also

Current Date and Time page 6, SQL Date and Time page 21.

Defaults, Setting

Chapter 1, PeopleCode Built-in Functions, SetDefault , page 415

Chapter 1, PeopleCode Built-in Functions, SetDefaultAll , page 416

Chapter 1, PeopleCode Built-in Functions, SetDefaultNext , page 417

Chapter 1, PeopleCode Built-in Functions, SetDefaultNextRel , page 417

Chapter 1, PeopleCode Built-in Functions, SetDefaultPrior , page 418

Chapter 1, PeopleCode Built-in Functions, SetDefaultPriorRel , page 418

Double-Byte Characters

Chapter 1, PeopleCode Built-in Functions, CharType , page 64

Chapter 1, PeopleCode Built-in Functions, ContainsCharType , page 75

Chapter 1, PeopleCode Built-in Functions, ContainsOnlyCharType , page 77

Chapter 1, PeopleCode Built-in Functions, ConvertChar , page 79

Chapter 1, PeopleCode Built-in Functions, DBCSTrim , page 129

Effective Date and Effective Sequence

Chapter 1, PeopleCode Built-in Functions, CurrEffDt , page 114

Chapter 1, PeopleCode Built-in Functions, CurrEffRowNum , page 114

Chapter 1, PeopleCode Built-in Functions, CurrEffSeq , page 115

Chapter 1, PeopleCode Built-in Functions, NextEffDt , page 331

Chapter 1, PeopleCode Built-in Functions, NextRelEffDt , page 331

Chapter 1, PeopleCode Built-in Functions, PriorEffDt , page 347

Chapter 1, PeopleCode Built-in Functions, PriorRelEffDt , page 348

Chapter 1, PeopleCode Built-in Functions, SetDefaultNext , page 417

Chapter 1, PeopleCode Built-in Functions, SetDefaultNextRel , page 417

Chapter 1, PeopleCode Built-in Functions, SetDefaultPrior , page 418

Chapter 1, PeopleCode Built-in Functions, SetDefaultPriorRel , page 418

Chapter 2, Meta-SQL , %EffDtCheck , page 527

Email

Chapter 1, PeopleCode Built-in Functions, AddEmailAddress , page 35

Chapter 1, PeopleCode Built-in Functions, ChangeEmailAddress , page 63

Chapter 1, PeopleCode Built-in Functions, DeleteEmailAddress , page 137

Chapter 1, PeopleCode Built-in Functions, MarkPrimaryEmailAddress , page 317

Environment

Chapter 1, PeopleCode Built-in Functions, ExpandEnvVar , page 178

Chapter 1, PeopleCode Built-in Functions, GetCwd , page 241

Chapter 1, PeopleCode Built-in Functions, GetEnv , page 241

Exceptions

Chapter 1, PeopleCode Built-in Functions, CreateException , page 97

Chapter 1, PeopleCode Built-in Functions, Throw , page 457

Chapter 1, PeopleCode Built-in Functions, Try , page 482

Executable Files, Running

Chapter 1, PeopleCode Built-in Functions, Exec , page 171

Chapter 1, PeopleCode Built-in Functions, WinExec , page 503

See Also

Object page 18.

Files

Chapter 1, PeopleCode Built-in Functions, CreateDirectory , page 96

Chapter 1, PeopleCode Built-in Functions, FileExists , page 184

Chapter 1, PeopleCode Built-in Functions, FindFiles , page 189

Chapter 1, PeopleCode Built-in Functions, GetFile , page 243

Chapter 1, PeopleCode Built-in Functions, RemoveDirectory , page 367

See Also

Images page 11, Attachment page 2.

Financial

Chapter 1, PeopleCode Built-in Functions, AccruableDays , page 27

Chapter 1, PeopleCode Built-in Functions, AccrualFactor , page 28

Chapter 1, PeopleCode Built-in Functions, BlackScholesCall , page 46

Chapter 1, PeopleCode Built-in Functions, BlackScholesPut , page 47

Chapter 1, PeopleCode Built-in Functions, BootstrapYTMs , page 48

Chapter 1, PeopleCode Built-in Functions, ConvertRate , page 86

Chapter 1, PeopleCode Built-in Functions, CubicSpline , page 112

Chapter 1, PeopleCode Built-in Functions, HermiteCubic , page 285

Chapter 1, PeopleCode Built-in Functions, HistVolatility , page 290

Chapter 1, PeopleCode Built-in Functions, LinearInterp , page 314

Grids

Chapter 1, PeopleCode Built-in Functions, GetGrid , page 246

Images

Chapter 1, PeopleCode Built-in Functions, DeleteImage , page 138

Chapter 1, PeopleCode Built-in Functions, GetImageExtents , page 249

Chapter 1, PeopleCode Built-in Functions, InsertImage , page 295

See Also

Attachment page 2, Files page 10.

Import Manager

Chapter 3, System Variables , %ExternalAuthInfo , page 568

Integration Broker

Chapter 1, PeopleCode Built-in Functions, EncryptNodePswd , page 160

Chapter 1, PeopleCode Built-in Functions, FindCodeSetValues , page 187

Chapter 1, PeopleCode Built-in Functions, NodeDelete , page 332

Chapter 1, PeopleCode Built-in Functions, NodeRename , page 333

Chapter 1, PeopleCode Built-in Functions, NodeSaveAs , page 333

Chapter 1, PeopleCode Built-in Functions, PingNode , page 346

Chapter 1, PeopleCode Built-in Functions, Transform , page 476

Internet

Chapter 1, PeopleCode Built-in Functions, CreateSOAPDoc , page 109

Chapter 1, PeopleCode Built-in Functions, EncodeURL , page 157

Chapter 1, PeopleCode Built-in Functions, EncodeURLForQueryString , page 158

Chapter 1, PeopleCode Built-in Functions, EscapeHTML , page 167

Chapter 1, PeopleCode Built-in Functions, EscapeJavascriptString , page 168

Chapter 1, PeopleCode Built-in Functions, EscapeWML , page 169

Chapter 1, PeopleCode Built-in Functions, GenerateActGuideContentUrl , page 198

Chapter 1, PeopleCode Built-in Functions, GenerateActGuidePortalUrl , page 199

Chapter 1, PeopleCode Built-in Functions, GenerateActGuideRelativeUrl , page 200

Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentRelURL , page 201

Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentURL , page 203

Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalRelURL , page 205

Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalURL , page 207

Chapter 1, PeopleCode Built-in Functions, GenerateComponentRelativeURL , page 209

Chapter 1, PeopleCode Built-in Functions, GenerateExternalPortalURL , page 211

Chapter 1, PeopleCode Built-in Functions, GenerateExternalRelativeURL , page 212

Chapter 1, PeopleCode Built-in Functions, GenerateHomepagePortalURL , page 213

Chapter 1, PeopleCode Built-in Functions, GenerateHomepageRelativeURL , page 214

Chapter 1, PeopleCode Built-in Functions, GenerateQueryContentURL , page 218

Chapter 1, PeopleCode Built-in Functions, GenerateQueryPortalURL , page 220

Chapter 1, PeopleCode Built-in Functions, GenerateQueryRelativeURL , page 221

Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentRelURL , page 223

Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentURL , page 224

Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalRelURL , page 226

Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalURL , page 227

Chapter 1, PeopleCode Built-in Functions, GenerateScriptRelativeURL , page 229

Chapter 1, PeopleCode Built-in Functions, GenerateTree , page 230

Chapter 1, PeopleCode Built-in Functions, GenerateWorklistPortalURL , page 231

Chapter 1, PeopleCode Built-in Functions, GenerateWorklistRelativeURL , page 232

Chapter 1, PeopleCode Built-in Functions, GetChartURL , page 240

Chapter 1, PeopleCode Built-in Functions, GetHTMLText , page 248

Chapter 1, PeopleCode Built-in Functions, GetMethodNames , page 254

Chapter 1, PeopleCode Built-in Functions, GetURL , page 279

Chapter 1, PeopleCode Built-in Functions, Unencode , page 483

Chapter 1, PeopleCode Built-in Functions, ViewURL , page 498

Chapter 3, System Variables , %EmailAddress , page 568

Chapter 3, System Variables , %Request , page 576

Chapter 3, System Variables , %Response , page 576

Chapter 3, System Variables , %RunningInPortal , page 577

Java

Chapter 1, PeopleCode Built-in Functions, CreateJavaArray , page 98

Chapter 1, PeopleCode Built-in Functions, CreateJavaObject , page 99

Chapter 1, PeopleCode Built-in Functions, GetJavaClass , page 252

Language Preference and Locale

Chapter 1, PeopleCode Built-in Functions, SetLanguage , page 420

Chapter 3, System Variables , %IsMultiLanguageEnabled , page 569

Chapter 3, System Variables , %Language , page 569

Chapter 3, System Variables , %Language_Base , page 569

Chapter 3, System Variables , %Market , page 570

Language Constructs

Chapter 1, PeopleCode Built-in Functions, Break , page 49

Chapter 1, PeopleCode Built-in Functions, Component , page 74

Chapter 1, PeopleCode Built-in Functions, Evaluate , page 169

Chapter 1, PeopleCode Built-in Functions, Exit , page 176

Chapter 1, PeopleCode Built-in Functions, For , page 192

Chapter 1, PeopleCode Built-in Functions, Function , page 196

Chapter 1, PeopleCode Built-in Functions, Global , page 281

Chapter 1, PeopleCode Built-in Functions, If , page 292

Chapter 1, PeopleCode Built-in Functions, Local , page 315

Chapter 1, PeopleCode Built-in Functions, Repeat , page 372

Chapter 1, PeopleCode Built-in Functions, Return , page 376

Chapter 1, PeopleCode Built-in Functions, Throw , page 457

Chapter 1, PeopleCode Built-in Functions, Try , page 482

Chapter 1, PeopleCode Built-in Functions, While , page 502

Logical (Tests for Blank Values)

Chapter 1, PeopleCode Built-in Functions, All , page 41

Chapter 1, PeopleCode Built-in Functions, AllOrNone , page 42

Chapter 1, PeopleCode Built-in Functions, None , page 335

Chapter 1, PeopleCode Built-in Functions, OnlyOne , page 344

Chapter 1, PeopleCode Built-in Functions, OnlyOneOrNone , page 345

Mail

Chapter 1, PeopleCode Built-in Functions, SendMail , page 394

Chapter 3, System Variables , %EmailAddress , page 568

Math

Chapter 1, PeopleCode Built-in Functions, Abs , page 27

Chapter 1, PeopleCode Built-in Functions, Acos , page 29

Chapter 1, PeopleCode Built-in Functions, Asin , page 45

Chapter 1, PeopleCode Built-in Functions, Atan , page 46

Chapter 1, PeopleCode Built-in Functions, Cos , page 91

Chapter 1, PeopleCode Built-in Functions, Cot , page 92

Chapter 1, PeopleCode Built-in Functions, Degrees , page 134

Chapter 1, PeopleCode Built-in Functions, Exp , page 176

Chapter 1, PeopleCode Built-in Functions, Fact , page 179

Chapter 1, PeopleCode Built-in Functions, GetNextNumber , page 257

Chapter 1, PeopleCode Built-in Functions, GetNextNumberWithGaps , page 259

Chapter 1, PeopleCode Built-in Functions, Idiv , page 291

Chapter 1, PeopleCode Built-in Functions, Int , page 298

Chapter 1, PeopleCode Built-in Functions, Integer , page 299

Chapter 1, PeopleCode Built-in Functions, Ln , page 315

Chapter 1, PeopleCode Built-in Functions, Log10 , page 316

Chapter 1, PeopleCode Built-in Functions, Max , page 319

Chapter 1, PeopleCode Built-in Functions, Min , page 326

Chapter 1, PeopleCode Built-in Functions, Mod , page 327

Chapter 1, PeopleCode Built-in Functions, Product , page 349

Chapter 1, PeopleCode Built-in Functions, Radians , page 356

Chapter 1, PeopleCode Built-in Functions, Rand , page 357

Chapter 1, PeopleCode Built-in Functions, Round , page 381

Chapter 1, PeopleCode Built-in Functions, Sign , page 438

Chapter 1, PeopleCode Built-in Functions, Sin , page 439

Chapter 1, PeopleCode Built-in Functions, Sqrt , page 447

Chapter 1, PeopleCode Built-in Functions, Tan , page 456

Chapter 1, PeopleCode Built-in Functions, Truncate , page 481

Chapter 2, Meta-SQL , %Abs , page 518

Chapter 2, Meta-SQL , %DecDiv , page 525

Chapter 2, Meta-SQL , %DecMult , page 526

Chapter 2, Meta-SQL , %Round , page 548

Chapter 2, Meta-SQL , %Truncate , page 556

Menu Appearance

Chapter 1, PeopleCode Built-in Functions, CheckMenuItem, page 67

Chapter 1, PeopleCode Built-in Functions, DisableMenuItem , page 146

Chapter 1, PeopleCode Built-in Functions, EnableMenuItem , page 157

Chapter 1, PeopleCode Built-in Functions, HideMenuItem , page 287

Chapter 1, PeopleCode Built-in Functions, UnCheckMenuItem, page 483

Message Catalog and Message Display

Chapter 1, PeopleCode Built-in Functions, Error , page 165

Chapter 1, PeopleCode Built-in Functions, MessageBox , page 319

Chapter 1, PeopleCode Built-in Functions, MsgGet , page 328

Chapter 1, PeopleCode Built-in Functions, MsgGetExplainText , page 329

Chapter 1, PeopleCode Built-in Functions, MsgGetText , page 330

Chapter 1, PeopleCode Built-in Functions, Warning , page 499

Chapter 1, PeopleCode Built-in Functions, WinMessage , page 505

Messages

Chapter 1, PeopleCode Built-in Functions, AddSystemPauseTimes , page 37

Chapter 1, PeopleCode Built-in Functions, CreateMessage , page 100

Chapter 1, PeopleCode Built-in Functions, DeleteSystemPauseTimes , page 143

Chapter 1, PeopleCode Built-in Functions, GetMessage , page 255

Chapter 1, PeopleCode Built-in Functions, GetSyncLogData , page 277

Chapter 1, PeopleCode Built-in Functions, IsMessageActive , page 307

Chapter 1, PeopleCode Built-in Functions, ReturnToServer , page 377

Chapter 1, PeopleCode Built-in Functions, PingNode , page 346

Chapter 1, PeopleCode Built-in Functions, SetChannelStatus , page 401

Chapter 3, System Variables , %MaxMessageSize , page 570

See Also

XML page 26.

Mobile

Chapter 1, PeopleCode Built-in Functions, GenerateMobileTree, page 215

Chapter 1, PeopleCode Built-in Functions, TransferMobilePage, page 467

Chapter 3, System Variables , %CompIntfcName , page 565

Chapter 3, System Variables , %DeviceType, page 567

Chapter 3, System Variables , %MobilePage, page 571

Chapter 3, System Variables , %Session , page 577

Chapter 3, System Variables , %ThisMobileObject, page 580

Modal Transfers

Chapter 1, PeopleCode Built-in Functions, DoModalComponent , page 150

Chapter 1, PeopleCode Built-in Functions, EndModalComponent , page 162

Chapter 1, PeopleCode Built-in Functions, IsModalComponent , page 308

See Also

Secondary Pages page 20, Transfers page 23.

MultiChannel Framework

Chapter 1, PeopleCode Built-in Functions, DeQueue , page 144

Chapter 1, PeopleCode Built-in Functions, EnQueue , page 163

Chapter 1, PeopleCode Built-in Functions, Forward , page 193

Chapter 1, PeopleCode Built-in Functions, InitChat , page 293

Chapter 1, PeopleCode Built-in Functions, NotifyQ , page 336

Object-Based

Chapter 1, PeopleCode Built-in Functions, CreateArray , page 92

Chapter 1, PeopleCode Built-in Functions, CreateArrayRept , page 94

Chapter 1, PeopleCode Built-in Functions, CreateException , page 97

Chapter 1, PeopleCode Built-in Functions, CreateJavaArray , page 98

Chapter 1, PeopleCode Built-in Functions, CreateJavaObject , page 99

Chapter 1, PeopleCode Built-in Functions, CreateMessage , page 100

Chapter 1, PeopleCode Built-in Functions, CreateObject , page 101

Chapter 1, PeopleCode Built-in Functions, CreateObjectArray , page 103

Chapter 1, PeopleCode Built-in Functions, CreateProcessRequest , page 104

Chapter 1, PeopleCode Built-in Functions, CreateRecord , page 105

Chapter 1, PeopleCode Built-in Functions, CreateRowset , page 107

Chapter 1, PeopleCode Built-in Functions, CreateSOAPDoc , page 109

Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109

Chapter 1, PeopleCode Built-in Functions, CreateXmlDoc , page 111

Chapter 1, PeopleCode Built-in Functions, DeleteSQL , page 141

Chapter 1, PeopleCode Built-in Functions, FetchSQL , page 180

Chapter 1, PeopleCode Built-in Functions, GetAESection , page 233

Chapter 1, PeopleCode Built-in Functions, GetChart , page 239

Chapter 1, PeopleCode Built-in Functions, GetChartURL , page 240

Chapter 1, PeopleCode Built-in Functions, GetCwd , page 241

Chapter 1, PeopleCode Built-in Functions, GetField , page 242

Chapter 1, PeopleCode Built-in Functions, GetFile , page 243

Chapter 1, PeopleCode Built-in Functions, GetGrid , page 246

Chapter 1, PeopleCode Built-in Functions, GetHTMLText , page 248

Chapter 1, PeopleCode Built-in Functions, GetInterlink , page 250

Chapter 1, PeopleCode Built-in Functions, GetLevel0 , page 253

Chapter 1, PeopleCode Built-in Functions, GetJavaClass , page 252

Chapter 1, PeopleCode Built-in Functions, GetMessage , page 255

Chapter 1, PeopleCode Built-in Functions, GetMessageXmlDoc , page 255

Chapter 1, PeopleCode Built-in Functions, GetMessageInstance , page 255

Chapter 1, PeopleCode Built-in Functions, GetPubContractInstance , page 263

Chapter 1, PeopleCode Built-in Functions, GetRecord , page 265

Chapter 1, PeopleCode Built-in Functions, GetRow , page 267

Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268

Chapter 1, PeopleCode Built-in Functions, GetSession , page 269

Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271

Chapter 1, PeopleCode Built-in Functions, GetSubContractInstance , page 274

Chapter 1, PeopleCode Built-in Functions, ObjectDoMethod , page 340

Chapter 1, PeopleCode Built-in Functions, ObjectDoMethodArray , page 341

Chapter 1, PeopleCode Built-in Functions, ObjectGetProperty , page 342

Chapter 1, PeopleCode Built-in Functions, ObjectSetProperty , page 343

Chapter 1, PeopleCode Built-in Functions, ReturnToServer , page 377

Chapter 1, PeopleCode Built-in Functions, Split , page 441

Chapter 1, PeopleCode Built-in Functions, StoreSQL , page 450

Object

Chapter 1, PeopleCode Built-in Functions, CreateObject , page 101

Chapter 1, PeopleCode Built-in Functions, CreateObjectArray , page 103

Chapter 1, PeopleCode Built-in Functions, ObjectDoMethod , page 340

Chapter 1, PeopleCode Built-in Functions, ObjectDoMethodArray , page 341

Chapter 1, PeopleCode Built-in Functions, ObjectGetProperty , page 342

Chapter 1, PeopleCode Built-in Functions, ObjectSetProperty , page 343

Page

Chapter 1, PeopleCode Built-in Functions, GetPage , page 262

Page Control Appearance

Chapter 1, PeopleCode Built-in Functions, GetImageExtents , page 249

Chapter 1, PeopleCode Built-in Functions, Gray , page 282

Chapter 1, PeopleCode Built-in Functions, Hide , page 285

Chapter 1, PeopleCode Built-in Functions, HideRow , page 287

Chapter 1, PeopleCode Built-in Functions, HideScroll , page 289

Chapter 1, PeopleCode Built-in Functions, IsHidden , page 304

Chapter 1, PeopleCode Built-in Functions, SetCursorPos , page 404

Chapter 1, PeopleCode Built-in Functions, SetLabel , page 419

Chapter 1, PeopleCode Built-in Functions, Ungray , page 484

Chapter 1, PeopleCode Built-in Functions, Unhide , page 486

Chapter 1, PeopleCode Built-in Functions, UnhideRow , page 487

Chapter 1, PeopleCode Built-in Functions, UnhideScroll , page 488

Personalizations

Chapter 1, PeopleCode Built-in Functions, GetUserOption , page 280

Chapter 1, PeopleCode Built-in Functions, SetUserOption , page 437

Process Scheduler

Chapter 1, PeopleCode Built-in Functions, CreateProcessRequest , page 104

Chapter 1, PeopleCode Built-in Functions, GetNextProcessInstance , page 261

Chapter 1, PeopleCode Built-in Functions, SetupScheduleDefnItem , page 437

Remote Call

Chapter 1, PeopleCode Built-in Functions, DoSaveNow , page 155

Chapter 1, PeopleCode Built-in Functions, RemoteCall , page 364

Saving and Canceling

Chapter 1, PeopleCode Built-in Functions, DoCancel , page 147

Chapter 1, PeopleCode Built-in Functions, DoSave , page 154

Chapter 1, PeopleCode Built-in Functions, DoSaveNow , page 155

Chapter 1, PeopleCode Built-in Functions, WinEscape , page 502

Scroll Select

Chapter 1, PeopleCode Built-in Functions, RowFlush , page 383

Chapter 1, PeopleCode Built-in Functions, RowScrollSelect , page 384

Chapter 1, PeopleCode Built-in Functions, RowScrollSelectNew , page 386

Chapter 1, PeopleCode Built-in Functions, ScrollFlush , page 388

Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389

Chapter 1, PeopleCode Built-in Functions, ScrollSelectNew , page 392

Chapter 1, PeopleCode Built-in Functions, SortScroll , page 440

Search Dialog

Chapter 1, PeopleCode Built-in Functions, ClearSearchDefault , page 68

Chapter 1, PeopleCode Built-in Functions, ClearSearchEdit , page 69

Chapter 1, PeopleCode Built-in Functions, IsSearchDialog , page 310

Chapter 1, PeopleCode Built-in Functions, SetSearchDefault , page 428

Chapter 1, PeopleCode Built-in Functions, SetSearchDialogBehavior , page 429

Chapter 1, PeopleCode Built-in Functions, SetSearchEdit , page 430

Chapter 3, System Variables , %Mode , page 571

Secondary Pages

Chapter 1, PeopleCode Built-in Functions, DoModal , page 148

Chapter 1, PeopleCode Built-in Functions, EndModal , page 161

Chapter 1, PeopleCode Built-in Functions, IsModal , page 307

See Also

Modal Transfers page 16, Transfers page 23.

SQL

Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109

Chapter 1, PeopleCode Built-in Functions, DeleteSQL , page 141

Chapter 1, PeopleCode Built-in Functions, ExpandBindVar , page 177

Chapter 1, PeopleCode Built-in Functions, ExpandSqlBinds , page 178

Chapter 1, PeopleCode Built-in Functions, FetchSQL , page 180

Chapter 1, PeopleCode Built-in Functions, FlushBulkInserts , page 191

Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271

Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442

Chapter 1, PeopleCode Built-in Functions, StoreSQL , page 450

Chapter 2, Meta-SQL , %FirstRows , page 528

Chapter 2, Meta-SQL , %InsertSelect , page 529

Chapter 2, Meta-SQL , %InsertValues , page 532

Chapter 2, Meta-SQL , %Join , page 533

Chapter 2, Meta-SQL , %KeyEqual , page 535

Chapter 2, Meta-SQL , %KeyEqualNoEffDt , page 536

Chapter 2, Meta-SQL , %Like , page 537

Chapter 2, Meta-SQL , %LikeExact , page 539

Chapter 2, Meta-SQL , %List , page 541

Chapter 2, Meta-SQL , %ListBind , page 544

Chapter 2, Meta-SQL , %ListEqual , page 544

Chapter 2, Meta-SQL , %NoUppercase , page 545

Chapter 2, Meta-SQL , %OldKeyEqual , page 547

Chapter 2, Meta-SQL , %SQL , page 548

Chapter 3, System Variables , %SignonUserId , page 578

Chapter 2, Meta-SQL , %Table , page 551

Chapter 2, Meta-SQL , %UpdatePairs , page 558

See Also

Scroll Select page 19, Data Buffer Access page 6.

SQL Date and Time

Chapter 2, Meta-SQL , %DateAdd , page 521

Chapter 2, Meta-SQL , %DateDiff , page 521

Chapter 2, Meta-SQL , %DateIn, page 522

Chapter 2, Meta-SQL , %DateOut , page 523

Chapter 2, Meta-SQL , %DateTimeDiff , page 523

Chapter 2, Meta-SQL , %DateTimeIn , page 524

Chapter 2, Meta-SQL , %DateTimeOut , page 525

Chapter 2, Meta-SQL , %DTTM , page 526

Chapter 2, Meta-SQL , %TimeAdd , page 553

Chapter 2, Meta-SQL , %TextIn , page 553

Chapter 2, Meta-SQL , %TimeIn , page 554

Chapter 2, Meta-SQL , %TimeOut , page 555

See Also

Current Date and Time page 6, Date and Time page 7.

SQL Shortcuts

Chapter 2, Meta-SQL , %Delete , page 560

Chapter 2, Meta-SQL , %Insert , page 560

Chapter 2, Meta-SQL , %SelectAll , page 560

Chapter 2, Meta-SQL , %SelectDistinct , page 561

Chapter 2, Meta-SQL , %SelectByKey , page 562

Chapter 2, Meta-SQL , %SelectByKeyEffDt , page 562

Chapter 2, Meta-SQL , %Update , page 562

String

Chapter 1, PeopleCode Built-in Functions, Clean , page 67

Chapter 1, PeopleCode Built-in Functions, Code , page 69

Chapter 1, PeopleCode Built-in Functions, Codeb , page 70

Chapter 1, PeopleCode Built-in Functions, DBCSTrim , page 129

Chapter 1, PeopleCode Built-in Functions, DBPatternMatch , page 129

Chapter 1, PeopleCode Built-in Functions, Exact , page 170

Chapter 1, PeopleCode Built-in Functions, ExpandBindVar , page 177

Chapter 1, PeopleCode Built-in Functions, ExpandEnvVar , page 178

Chapter 1, PeopleCode Built-in Functions, Find , page 185

Chapter 1, PeopleCode Built-in Functions, Findb , page 186

Chapter 1, PeopleCode Built-in Functions, GetHTMLText , page 248

Chapter 1, PeopleCode Built-in Functions, IsAlpha , page 300

Chapter 1, PeopleCode Built-in Functions, IsAlphaNumeric , page 300

Chapter 1, PeopleCode Built-in Functions, IsDigits , page 304

Chapter 1, PeopleCode Built-in Functions, Left , page 313

Chapter 1, PeopleCode Built-in Functions, Len , page 313

Chapter 1, PeopleCode Built-in Functions, Lenb , page 314

Chapter 1, PeopleCode Built-in Functions, Lower , page 316

Chapter 1, PeopleCode Built-in Functions, LTrim , page 317

Chapter 1, PeopleCode Built-in Functions, NumberToString , page 337

Chapter 1, PeopleCode Built-in Functions, Proper , page 352

Chapter 1, PeopleCode Built-in Functions, Replace , page 373

Chapter 1, PeopleCode Built-in Functions, Rept , page 373

Chapter 1, PeopleCode Built-in Functions, Right , page 381

Chapter 1, PeopleCode Built-in Functions, RTrim , page 387

Chapter 1, PeopleCode Built-in Functions, String , page 451

Chapter 1, PeopleCode Built-in Functions, Substitute , page 452

Chapter 1, PeopleCode Built-in Functions, Substring , page 452

Chapter 1, PeopleCode Built-in Functions, Substringb , page 453

Chapter 1, PeopleCode Built-in Functions, Upper , page 492

Chapter 2, Meta-SQL , %Abs , page 518

Chapter 2, Meta-SQL , %Substring , page 550

Chapter 2, Meta-SQL , %TrimSubstr , page 556

Chapter 2, Meta-SQL , %Upper , page 559

Subrecords

Chapter 4, Meta-HTML, page 583

TimeZone

Chapter 1, PeopleCode Built-in Functions, ConvertDatetimeToBase , page 85

Chapter 1, PeopleCode Built-in Functions, ConvertTimeToBase , page 87

Chapter 1, PeopleCode Built-in Functions, DateTimeToTimeZone , page 124

Chapter 1, PeopleCode Built-in Functions, FormatDateTime , page 192

Chapter 1, PeopleCode Built-in Functions, IsDaylightSavings , page 302

Chapter 1, PeopleCode Built-in Functions, TimeToTimeZone , page 460

Chapter 1, PeopleCode Built-in Functions, TimeZoneOffset , page 461

Chapter 3, System Variables , %ClientTimeZone , page 564

Chapter 3, System Variables , %ServerTimeZone , page 577

Trace Control

Chapter 1, PeopleCode Built-in Functions, SetTracePC , page 432

Chapter 1, PeopleCode Built-in Functions, SetTraceSQL , page 435

Transfers

Chapter 1, PeopleCode Built-in Functions, AddKeyListItem , page 36

Chapter 1, PeopleCode Built-in Functions, ClearKeyList , page 67

Chapter 1, PeopleCode Built-in Functions, SetNextPage , page 422

Chapter 1, PeopleCode Built-in Functions, Transfer , page 463

Chapter 1, PeopleCode Built-in Functions, TransferNode , page 468

Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475

Chapter 1, PeopleCode Built-in Functions, TransferPortal , page 471

See Also

Modal Transfers page 16, Secondary Pages page 20.

Type Checking

Chapter 1, PeopleCode Built-in Functions, IsAlpha , page 300

Chapter 1, PeopleCode Built-in Functions, IsAlphaNumeric , page 300

Chapter 1, PeopleCode Built-in Functions, IsDate , page 301

Chapter 1, PeopleCode Built-in Functions, IsDateTime , page 302

Chapter 1, PeopleCode Built-in Functions, IsDigits , page 304

Chapter 1, PeopleCode Built-in Functions, IsNumber , page 309

Chapter 1, PeopleCode Built-in Functions, IsTime , page 311

Chapter 1, PeopleCode Built-in Functions, Max , page 319

Chapter 1, PeopleCode Built-in Functions, Min , page 326

Chapter 1, PeopleCode Built-in Functions, NumberToString , page 337

User Information

Chapter 3, System Variables , %EmployeeId , page 568

Chapter 3, System Variables , %EmailAddress , page 568

Chapter 3, System Variables , %UserDescription , page 581

Chapter 3, System Variables , %UserId , page 581

User Security

Chapter 1, PeopleCode Built-in Functions, AllowEmplIdChg , page 43

Chapter 1, PeopleCode Built-in Functions, Decrypt , page 133

Chapter 1, PeopleCode Built-in Functions, Encrypt , page 159

Chapter 1, PeopleCode Built-in Functions, ExecuteRolePeopleCode , page 173

Chapter 1, PeopleCode Built-in Functions, ExecuteRoleQuery , page 174

Chapter 1, PeopleCode Built-in Functions, ExecuteRoleWorkflowQuery , page 175

Chapter 1, PeopleCode Built-in Functions, Hash , page 284

Chapter 1, PeopleCode Built-in Functions, IsMenuItemAuthorized , page 305

Chapter 1, PeopleCode Built-in Functions, IsUserInPermissionList , page 312

Chapter 1, PeopleCode Built-in Functions, IsUserInRole , page 312

Chapter 1, PeopleCode Built-in Functions, ReturnToServer , page 377

Chapter 1, PeopleCode Built-in Functions, SetAuthenticationResult , page 399

Chapter 1, PeopleCode Built-in Functions, SetPasswordExpired , page 424

Chapter 1, PeopleCode Built-in Functions, SwitchUser , page 454

Chapter 3, System Variables , %AuthenticationToken , page 564

Chapter 3, System Variables , %EmailAddress , page 568

Chapter 3, System Variables , %ExternalAuthInfo , page 568

Chapter 3, System Variables , %NavigatorHomePermissionList , page 571

Chapter 3, System Variables , %PasswordExpired , page 574

Chapter 3, System Variables , %PermissionLists , page 575

Chapter 3, System Variables , %PrimaryPermissionList , page 575

Chapter 3, System Variables , %ProcessProfilePermissionList , page 576

Chapter 3, System Variables , %PSAuthResult , page 576

Chapter 3, System Variables , %ResultDocument , page 576

Chapter 3, System Variables , %Roles , page 577

Chapter 3, System Variables , %RowSecurityPermissionList , page 577

Chapter 3, System Variables , %SignonUserId , page 578

Chapter 3, System Variables , %SignOnUserPswd , page 578

Chapter 3, System Variables , %UserId , page 581

Validation

Chapter 1, PeopleCode Built-in Functions, Error , page 165

Chapter 1, PeopleCode Built-in Functions, IsMenuItemAuthorized , page 305

Chapter 1, PeopleCode Built-in Functions, RevalidatePassword , page 379

Chapter 1, PeopleCode Built-in Functions, SetCursorPos , page 404

Chapter 1, PeopleCode Built-in Functions, SetReEdit , page 428

Chapter 1, PeopleCode Built-in Functions, Warning , page 499

Workflow

Chapter 1, PeopleCode Built-in Functions, GenerateActGuideContentUrl , page 198

Chapter 1, PeopleCode Built-in Functions, GenerateActGuidePortalUrl , page 199

Chapter 1, PeopleCode Built-in Functions, GenerateActGuideRelativeUrl , page 200

Chapter 1, PeopleCode Built-in Functions, GetWLFieldValue , page 281

Chapter 1, PeopleCode Built-in Functions, MarkWLItemWorked , page 318

Chapter 1, PeopleCode Built-in Functions, TriggerBusinessEvent , page 481

Chapter 3, System Variables , %BPName , page 564

Chapter 3, System Variables , %WLInstanceID , page 581

Chapter 3, System Variables , %WLName , page 581

XML

Chapter 1, PeopleCode Built-in Functions, CancelPubHeaderXmlDoc , page 61

Chapter 1, PeopleCode Built-in Functions, CancelPubXmlDoc , page 61

Chapter 1, PeopleCode Built-in Functions, CancelSubXmlDoc , page 62

Chapter 1, PeopleCode Built-in Functions, CreateSOAPDoc , page 109

Chapter 1, PeopleCode Built-in Functions, CreateXmlDoc , page 111

Chapter 1, PeopleCode Built-in Functions, GetMessageXmlDoc , page 255

Chapter 1, PeopleCode Built-in Functions, GetNRXmlDoc , page 261

Chapter 1, PeopleCode Built-in Functions, GetPubHeaderXmlDoc , page 263

Chapter 1, PeopleCode Built-in Functions, GetPubXmlDoc , page 264

Chapter 1, PeopleCode Built-in Functions, GetSubXmlDoc , page 276

Chapter 1, PeopleCode Built-in Functions, GetSyncLogData , page 277

Chapter 1, PeopleCode Built-in Functions, PublishXmlDoc , page 352

Chapter 1, PeopleCode Built-in Functions, ReSubmitPubHeaderXmlDoc , page 374

Chapter 1, PeopleCode Built-in Functions, ReSubmitPubXmlDoc , page 375

Chapter 1, PeopleCode Built-in Functions, ReSubmitSubXmlDoc , page 376

Chapter 1, PeopleCode Built-in Functions, ReValidateNRXmlDoc , page 379

Chapter 1, PeopleCode Built-in Functions, SyncRequestXmlDoc , page 455

Chapter 1, PeopleCode Built-in Functions, Transform , page 476

Chapter 1, PeopleCode Built-in Functions, UpdateXmlDoc , page 492

PeopleCode Built-in Functions and Language Constructs

The following are the PeopleCode Built-In functions.

Abs

Syntax

Abs(*x*)

Description

Abs returns a decimal value equal to the absolute value of a number *x*.

Example

The example returns the absolute value of the difference between &NUM_1 and &NUM_2:

```
&RESULT = Abs(&NUM_1 - &NUM_2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Sign , page 438, Chapter 2, Meta-SQL , %Abs , page 518.

AccruableDays

Syntax

AccruableDays(*StartDate*, *EndDate*, *Accrual_Conv*)

Description

The **AccruableDays** function returns the number of days during which interest can accrue in a given range of time according to the *Accrual_Conv* parameter.

Parameters

Parameter	Description
StartDate	The beginning of the time period for determining the accrual. This parameter takes a date value.
EndDate	The end of the time period for determining the accrual. This parameter takes a date value.
Accrual_Conv	The accrual convention. This parameter takes either a number or a constant value. Values for this parameter are:

Numeric Value	Constant Value	Description
0	%Accrual_30DPM	30/360: all months 30 days long according to NASD rules for date truncation
1	%Accrual_30DPME	30E/360: all months 30 days long according to European rules for date truncation

Numeric Value	Constant Value	Description
2		30N/360: all months but February are 30 days long according to SIA rules
3	%Accrual_Fixed360	Act/360: months have variable number of days, but years have fixed 360 days
4	%Accrual_Fixed365	Act/365: months have variable number of days, buy years have fixed 365 days
5	%Accrual_ActualDPY	Act/Act: months and years have a variable number of days

Returns

An integer representing a number of days.

See Also

Chapter 1, PeopleCode Built-in Functions, AccrualFactor , page 28.

AccrualFactor

Syntax

AccrualFactor(*StartDate, EndDate, Accrual_Conv*)

Description

The **AccrualFactor** function computes a factor that's equal to the number of years of interest accrued during a date range, according to *Accrual_Conv* parameter.

Parameters

Parameter	Description
StartDate	The beginning of the time period for determining the accrual. This parameter takes a date value.
EndDate	The end of the time period for determining the accrual. This parameter takes a date value.
Accrual_Conv	The accrual convention. This parameter takes either a number or constant value. Values for this parameter are:

Numeric Value	Constant Value	Description
0	%Accrual_30DPM	30/360: all months 30 days long according to NASD rules for date truncation
1	%Accrual_30DPME	30E/360: all months 30 days long according to European rules for date truncation
2		30N/360: all months but February are 30 days long according to SIA rules
3	%Accrual_Fixed360	Act/360: months have variable number of days, but years have fixed 360 days
4	%Accrual_Fixed365	Act/365: months have variable number of days, buy years have fixed 365 days

Numeric Value	Constant Value	Description
5	%Accrual_ActualDPY	Act/Act: months and years have a variable number of days

Returns

A floating point number representing a number of years.

See Also

Chapter 1, PeopleCode Built-in Functions, AccruableDays , page 27.

Acos

Syntax

Acos(*value*)

Description

The **Acos** function calculates the arccosine of the given value, that is, the size of the angle whose cosine is that value.

Parameters

Parameter	Description
value	Any real number between **-1.00** and **1.00** inclusive, the range of valid cosine values. If the input value is outside this range, an error message appears at runtime ("Decimal arithmetic error occurred. (2,110)"). Adjust your code to provide a valid input value.

Returns

A value in radians between **0** and **pi**.

Example

The following example returns the size in radians of the angle whose cosine is **0.5**:

```
&MY_ANGLE = Acos(0.5);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Asin , page 45, Chapter 1, PeopleCode Built-in Functions, Atan , page 46, Chapter 1, PeopleCode Built-in Functions, Cos , page 91, Chapter 1, PeopleCode Built-in Functions, Cot , page 92, Chapter 1, PeopleCode Built-in Functions, Degrees , page 134, Chapter 1, PeopleCode Built-in Functions, Radians , page 356, Chapter 1, PeopleCode Built-in Functions, Sin , page 439, Chapter 1, PeopleCode Built-in Functions, Tan , page 456.

ActiveRowCount

Syntax

ActiveRowCount(*Scrollpath*)

Where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD**.*target_recname*

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

ActiveRowCount returns the number of active (non-deleted) rows for a specified scroll area in the active page.

Note. This function remains for backward compatibility only. Use the **ActiveRowCount** Rowset class property instead.

ActiveRowCount is often used to get a limiting value for a For statement. This enables you to loop through the active rows of a scroll area, performing an operation on each active row. Rows that have been marked as deleted are not affected in a **For** loop delimited by ActiveRowCount. If you want to loop through all the rows of a scroll area, including deleted rows, use TotalRowCount.

Use ActiveRowCount with CurrentRowNumber to determine whether the user is on the last row of a record.

See Also

Chapter 27, Rowset Class , ActiveRowCount, page 1502, *PeopleCode Developer's Guide*, Data Buffer Access.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.

Returns

Returns a Number value equal to the total active (non-deleted) rows in the specified scroll area in the active page.

Example

In this example **ActiveRowCount** is used to delimit a **For** loop through a level-one scroll:

```
&CURRENT_L1 = CurrentRowNumber(1);
&ACTIVE_L2 = ActiveRowCount(RECORD.ASSIGNMENT, &CURRENT_L1, RECORD.ASGN_HOME_HOST);
&HOME_HOST = FetchValue(RECORD.ASSIGNMENT, &CURRENT_L1,
ASGN_HOME_HOST.HOME_HOST, 1);
If All(&HOME_HOST) Then
   For &I = 1 To &ACTIVE_L2
      DeleteRow(RECORD.ASSIGNMENT, &CURRENT_L1, RECORD.ASGN_HOME_HOST, 1);
   End-For;
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CurrentRowNumber , page 116, Chapter 1, PeopleCode Built-in Functions, TotalRowCount , page 462, Chapter 1, PeopleCode Built-in Functions, For , page 192.

AddAttachment

Syntax

AddAttachment(*URLDestination*, *DirAndFilename*, *FileType*, *UserFile*, *MaxSize*)

where *URLDestination* can have *one* of the following forms:
URL.*URLname*

Or a string URL, such as
ftp://user:password@ftp.ps.com/

Description

The **AddAttachment** function enables you to transfer a file from one server to another. You could use this to associate a file with a row in a record, or with a page of data.

Note. If the specified subdirectories don't exist this function tries to create them.

File Name Considerations

When the file is transferred, the following characters are replaced with an underscore:

- space
- semicolon
- plus sign
- percent sign
- ampersand
- apostrophe
- exclamation point
- @ sign
- pound sign
- dollar sign

Restrictions on Use in PeopleCode Events

AddAttachment is a "think-time" function, which means that it should not be used in any of the following PeopleCode events:

- SavePreChange
- SavePostChange
- Workflow
- RowSelect
- Any PeopleCode event that initiates as a result of a Select or SelectNew method, or any of the ScrollSelect functions.

If you want to transfer files in a "non-interactive" mode, with functions that aren't think-time, see the GetAttachment and PutAttachment functions.

Security Considerations

The *FileType* parameter is only a suggestion. Specifying it does *not* enforce specific file types. If you do not want specific file types (such as .exe or .bat) you must write PeopleCode to discern the name and file type of the attached file and delete it.

When the end user views attachments using the ViewAttachment function, some browsers treat documents as HTML *regardless of file extension*, and thus execute embedded java script. You may want to write a PeopleCode program to only allow specific file extensions to be viewed.

See Also

Chapter 1, PeopleCode Built-in Functions, GetAttachment , page 234, Chapter 1, PeopleCode Built-in Functions, PutAttachment , page 353, Chapter 27, Rowset Class , Select, page 1495, Chapter 27, Rowset Class , SelectNew, page 1498, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Think-Time Functions.

Parameters

Parameter	Description
URLDestination	A reference to a URL. This can be either a URL name, in the form URL.*URLName*, or a string. This is where the file is transferred to.
DirAndFileName	A directory and filename. This is appended to the *URLDestination* to make up the full URL where the file is transferred to. **Note.** The *URLDestination* requires "/" (forward) slashes. Because the *DirAndFileName* parameter is appended to the URL, it also requires only "/" slashes. "\" (backward) slashes are *not* supported for either the *URLDestination* or the *DirAndFileName* parameter.
FileType	A string to use as a suggestion for the extension file type, such as ".doc", ".gif", ".properties" and so on. The user may still type in a filename of a different extension. The value given for this parameter is *not* enforced. Passing in a null string (that is, two double-quotes with no space ("")) invokes all files (*.*).
UserFile	The name of the file on the source system.
MaxSize	Specify, in kilobytes, the maximum size of the attachment. If you specify if 0, it indicates “no limit,” so any filesize can be added. This includes files with zero bytes. This is the default value. **Note.** *MaxSize* is checked before the file is transferred to the web server. If the file exceeds the value specified with the *MaxSize* parameter, it is not transferred, and the system issues an error message.

Returns

You can check for either an integer or a constant value:

Numeric Value	Constant Value	Description
0	%Attachment_Success	File was transferred successfully.
1	%Attachment_Failed	File was not successfully transferred.
2	%Attachment_Cancelled	File transfer didn’t complete because the operation was canceled by the user.

Numeric Value	Constant Value	Description
3	%Attachment_FileTransferFailed	File transfer didn't succeed.
4	%Attachment_NoDiskSpaceAppServ	No disk space on the application server.
5	%Attachment_NoDiskSpaceWebServ	No disk space on the web server.
6	%Attachment_FileExceedsMaxSize	File exceeds maximum size.
7	%Attachment_DestSystNotFound	Cannot locate destination system for ftp.
8	%Attachment_DestSystFailedLogin	Unable to login to destination system for ftp.
9	%Attachment_FileNotFound	Cannot locate file.
10	%Attachment_DeleteFailed	Cannot delete file.

Example

The following example is how to specify a URL for use with AddAttachment, with encryption, de-referencing the parameter for use with AddAttachment:

The following is for creating the password:

```
If All(&OPERPSWD) Then
    OPERPSWD = Encrypt("some value", RTrim(LTrim(&WRKOPERPSWD)));
End-If;
```

The following code example shows how to access the password and enclose it in a URL:

```
SQLExec("select ftpid, operpswd, ftpaddress, ftpdirectory from PS_CDM_DIST_NODE⇒
 where distnodename = 'HRFTP'", &FTPID, &OPERPSWD,
&ftpaddress, &ftpdirectory);

If All(&OPERPSWD) Then
    &OPERPSWD = Decrypt("some value", RTrim(LTrim(&OPERPSWD)));
End-If;
```

Here is another example.

```
&uniquefilename = "";
&REC = GetRecord();
For &I = 1 To &REC.FieldCount
   &Fld = &REC.getfield(&I);
   If (&Fld.IsKey = True) Then
      If (&Fld.value = "" And
            &Fld.IsRequired = True) Then
         MessageBox(0, "File Attachment Status", 0, 0, "Please attach the file⇒
 after filling out the required fields on this page.");
         Return;
      Else
         &uniquefilename = &uniquefilename | &Fld.value;
      End-If;
   End-If;
End-For;
&temp = Substitute(&uniquefilename, " ", ""); /* delete spaces */
```

```
ATTACHSYSFILENAME = Left(&uniquefilename, 64);
/* make uniq part a max of 64 chars */

/* If you have created a subdirectory on the ftp server for this page,
prepend it here:

   ATTACHSYSFILENAME = "HR/ABSENCEHIST/" | ATTACHSYSFILENAME;

*/

&retcode = AddAttachment(URL.MYFTP, ATTACHSYSFILENAME, "", ATTACHUSERFILE, 0);

/* The actual name on the ftp file server is the following */

ATTACHSYSFILENAME = ATTACHSYSFILENAME | ATTACHUSERFILE;
&temp = Substitute(ATTACHSYSFILENAME, " ", ""); /* delete spaces */
ATTACHSYSFILENAME = &temp;

If (&retcode = %Attachment_Success) Then
   MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment succeeded");
   Hide(ATTACHADD);
   UnHide(ATTACHVIEW);
   UnHide(ATTACHDELETE);
End-If;
If (&retcode = %Attachment_Failed) Then
   MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed");
   ATTACHUSERFILE = "";
   ATTACHSYSFILENAME = "";
End-If;
If (&retcode = %Attachment_Cancelled) Then
   MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment cancelled");
   ATTACHUSERFILE = "";
   ATTACHSYSFILENAME = "";
End-If;
If (&retcode = %Attachment_FileTransferFailed) Then
   MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: File⇒
 Transfer did not succeed");
   ATTACHUSERFILE = "";
   ATTACHSYSFILENAME = "";
End-If;
/* following error message only in iclient mode */
If (&retcode = %Attachment_NoDiskSpaceAppServ) Then
   MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: No disk⇒
 space on the app server");
   ATTACHUSERFILE = "";
   ATTACHSYSFILENAME = "";
End-If;
/* following error message only in iclient mode */
If (&retcode = %Attachment_NoDiskSpaceWebServ) Then
```

```
   MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: No disk⇒
 space on the web server");
   ATTACHUSERFILE = "";
   ATTACHSYSFILENAME = "";
End-If;
If (&retcode = %Attachment_FileExceedsMaxSize) Then
   MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: File⇒
 exceeds the max size");
   ATTACHUSERFILE = "";
   ATTACHSYSFILENAME = "";
End-If;
If (&retcode = %Attachment_DestSystNotFound) Then
   MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: Cannot⇒
 locate destination system for ftp");
   ATTACHUSERFILE = "";
   ATTACHSYSFILENAME = "";
End-If;
If (&retcode = %Attachment_DestSysFailedLogin) Then
   MessageBox(0, "File Attachment Status", 0, 0, "AddAttachment failed: Unable to⇒
 login into destination system for ftp");
   ATTACHUSERFILE = "";
   ATTACHSYSFILENAME = "";
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, DeleteAttachment , page 135, Chapter 1, PeopleCode Built-in Functions, GetAttachment , page 234, Chapter 1, PeopleCode Built-in Functions, PutAttachment , page 353, Chapter 1, PeopleCode Built-in Functions, ViewAttachment , page 494, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Understanding File Attachment Architecture.

AddEmailAddress

Syntax

AddEmailAddress(*Type*, *Address* [, *Primary*])

Description

Use the **AddEmailAddress** function to add an email address for the current user. You can only add *one* email address of a specific type for a user. If you try to add an email address for a type that is already associated with an email address, you receive an error.

Parameters

Parameter	Description
Type	Specify the type of email address being added. This parameter takes a string value. The valid values are:

Value	Description
BB	Blackberry email address
BUS	Business email address
HOME	Home email address
OTH	Other email address
WORK	Work email address

Parameter	Description
Address	Specify the email address that you want to add as a string.
Primary	Specify whether this email address is the primary address for the user. This parameter takes a Boolean value: True, this email address is the primary email address, False otherwise. If not specified, the default is False.

Returns

None.

See Also

Chapter 1, PeopleCode Built-in Functions, ChangeEmailAddress , page 63, Chapter 1, PeopleCode Built-in Functions, DeleteEmailAddress , page 137, Chapter 1, PeopleCode Built-in Functions, MarkPrimaryEmailAddress , page 317.

AddKeyListItem

Syntax

```
AddKeyListItem(field, value)
```

Description

AddKeyListItem adds a new key field and its value to the current list of keys. It enables PeopleCode to help users navigate through related pages without being prompted for key values. A common use of **AddKeyListItem** is to add a field to a key list and then transfer to a page which uses that field as a key.

Parameters

Parameter	Description
field	The field to add to the key list.
value	The value of the added key field used in the search.

Returns

Returns a Boolean value indicating whether it completed successfully.

Example

The following example creates a key list using **AddKeyListItem** and transfers the user to a page named VOUCHER_INQUIRY_FS.

```
AddKeyListItem(VNDR_INQ_VW_FS.BUSINESS_UNIT, ASSET_ACQ_DET.BUSINESS_UNIT_AP);
AddKeyListItem(VNDR_INQ_VW_FS.VOUCHER_ID, ASSET_ACQ_DET.VOUCHER_ID);
TransferPage("VOUCHER_INQUIRY_FS");
```

See Also

Chapter 1, PeopleCode Built-in Functions, ClearKeyList , page 67, Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475, Chapter 1, PeopleCode Built-in Functions, Transfer , page 463.

AddSystemPauseTimes

Syntax

AddSystemPauseTimes(*StartDay*, *StartTime*, *EndDay*, *EndTime*)

Description

The **AddSystemPauseTimes** function enables you to set when pause times occur on your system by adding a row to the system pause-times tables.

This function is used in the PeopleCode for the Message Monitor. Pause times are set up in the Message Monitor.

Parameters

Parameter	Description
StartDay	Specify a number from 0-6. Values are:

Value	Description
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

Parameter	Description
StartTime	Specify a time, in seconds, since midnight.
EndDay	Specify a number from 0-6. Values are:

Value	Description
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

Parameter	Description
EndTime	Specify a time, in seconds, since midnight.

Returns

A Boolean value: True if the system pause time specified was added, False otherwise.

Example

```
Declare Function SetTime PeopleCode REFRESH_BTN FieldFormula;

Component Boolean &spt_changed;

Function GetSecond(&time) Returns number ;
   Return Hour(&time) * 3600 + Minute(&time) * 60 + Second(&time);
End-Function;

/* initialize; */

STARTDAY = "0";
AMM_STARTTIME = SetTime(0);
ENDDAY = "0";
AMM_ENDTIME = SetTime(0);

If DoModal(Panel.AMM_ADD_SPTIMES, MsgGetText(117, 13, ""), - 1, - 1) = 1 Then
   If AddSystemPauseTimes(Value(STARTDAY), GetSecond(AMM_STARTTIME), ⇒
Value(ENDDAY), GetSecond(AMM_ENDTIME)) Then
      &spt_changed = True;
      DoSave();
   Else
      MessageBox(16, MsgGetText(117, 13, ""), 117, 14, "");
   End-If;
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, DeleteSystemPauseTimes , page 143, *PeopleSoft Integration Broker*, Using Integration Broker Monitor.

AddToDate

Syntax

AddToDate(*date, num_years, num_months, num_days*)

Description

The **AddToDate** function adds the specified number of years, months, and days to the *date* provided.

Suppose, for example, that you want to find a date six years from now. You could not just multiply 6 times 365 and add the result to today's date, because of leap years. And, depending on the current year, there may be one or two leap years in the next six years. **AddToDate** takes care of this for you.

You can subtract from dates by passing the function negative numbers.

Parameters

Parameter	Description
date	The input date to be adjusted.
num_years	The number of years by which to adjust the specified *date*. *num_years* can be a negative number.
num_months	The number of months by which to adjust the specified *date*. This parameter can be a negative number.
num_days	The number of days by which to adjust the specified *date*. This parameter can be a negative number.

Returns

Returns a Date value equal to the original date plus the number of years, months, and days passed to the function.

Example

The following example finds the date one year, three months, and 16 days after a field called BEGIN_DT:

```
AddToDate(BEGIN_DT, 1, 3, 16);
```

This example finds the date two months ago prior to BEGIN_DT:

```
AddToDate(BEGIN_DT, 0, -2, 0);
```

See Also

Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Day , page 127, Chapter 1, PeopleCode Built-in Functions, Days , page 128, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129, Chapter 1, PeopleCode Built-in Functions, Month , page 327, Chapter 1, PeopleCode Built-in Functions, Weekday , page 501.

AddToDateTime

Syntax

AddToDateTime(*datetime*, *years*, *months*, *days*, *hours*, *minutes*, *seconds*)

Description

The **AddToDateTime** function adds the specified number of years, months, days, hours, seconds, and minutes to the *datetime* provided. You can subtract from datetimes by passing the function negative numbers.

Parameters

Parameter	Description
datetime	The Datetime value to which you want to add.
years	An integer representing the number of years.
months	An integer representing the number of months to add to *datetime*.
days	An integer representing the number of days to add to *datetime*.
hours	An integer representing the number of hours to add to *datetime*.
minutes	An integer representing the number of minutes to add to *datetime*.
seconds	An integer representing the number of seconds to add to *datetime*.

Returns

Returns a Datetime value equal to the original date plus the number of years, months, days, hours, minutes, and seconds passed to the function.

Example

The following example postpones an interview scheduled in the INTRTime field by two days and two hours:

```
INTRTIME = AddToDateTime(INTRTIME, 0, 0, 2, 2, 0, 0);
```

See Also

Chapter 1, PeopleCode Built-in Functions, AddToTime , page 40, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, DateTimeValue , page 125, Chapter 1, PeopleCode Built-in Functions, TimeValue , page 461.

AddToTime

Syntax

AddToTime(*time*, *hours*, *minutes*, *seconds*)

Description

AddToTime adds *hours*, *minutes*, and *seconds* to *time*, and returns the result as a Time value. To subtract from *time*, use negative numbers for *hours*, *minutes*, and *seconds*. The resulting value is always adjusted such that it represents an hour less than 24 (a valid time of day.)

Parameters

Parameter	Description
time	A time value that you want to subtract from or add to.
hours	An integer representing the number of hours to add to *time*.
minutes	An integer representing the number of minutes to add to *time*.
seconds	An integer representing the number of seconds to add to *time*.

Returns

A Time value equal to time increased by the number of hours, minutes, and seconds passed to the function.

Example

Assume that a time, &BREAKTime, is 0:15:00. The following moves the time &BREAKTime back by one hour, resulting in 23:15:00:

```
&BREAKTime = AddToTime(&BREAKTime, -1, 0, 0);
```

See Also

Chapter 1, PeopleCode Built-in Functions, AddToDateTime , page 40, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, DateTimeValue , page 125, Chapter 1, PeopleCode Built-in Functions, TimeValue , page 461.

All

Syntax

All(*fieldlist*)

Where *fieldlist* is an arbitrary-length list of field names in the form:
[*recordname.*]*fieldname1* [, [*recordname.*]*fieldname2*] ...

Description

All checks to see if a field contains a value, or if all the fields in a list of fields contain values. If any of the fields are Null, then **All** returns False.

A blank character field, or a zero (0) numeric value in a required numeric field is considered a null value.

Related Functions

None Checks that a field or list of fields have no value.

AllOrNone Checks if either all the field parameters have values, or none of them have values. Use this in examples where if the user fills in one field, she must fill in all the other related values.

OnlyOne Checks if exactly one field in the set has a value. Use this when the user must fill in only one of a set of mutually exclusive fields.

OnlyOneOrNone Checks if no more than one field in the set has a value. Use this in examples when a set of fields is both optional and mutually exclusive; that is, if the user can put a value into one field in a set of fields, or leave them all empty.

Returns

Returns a Boolean value based on the values in *fieldlist*. **All** returns True if all of the specified fields have a value; it returns False if any one of the fields does not contain a value.

Example

All is commonly used in SaveEdit PeopleCode to ensure that a group of related fields are all entered. For example:

```
If All(RETURN_DT, BEGIN_DT) and
    8 * (RETURN_DT - BEGIN_DT) (DURATION_DAYS * 8 + DURATION_HOURS)
Then
    Warning MsgGet(1000, 1, "Duration of absence exceeds standard hours for number⇒
 of days absent.");
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, AllOrNone , page 42, Chapter 1, PeopleCode Built-in Functions, None , page 335, Chapter 1, PeopleCode Built-in Functions, OnlyOne , page 344, Chapter 1, PeopleCode Built-in Functions, OnlyOneOrNone , page 345, Chapter 1, PeopleCode Built-in Functions, SetDefault , page 415, Chapter 1, PeopleCode Built-in Functions, SetDefaultAll , page 416.

AllOrNone

Syntax

AllOrNone(*fieldlist*)

Where *fieldlist* is an arbitrary-length list of field references in the form:
[*recordname.*]*fieldname1* [, [*recordname.*]*fieldname2*] ...

Description

AllOrNone takes a list of fields and returns True if either of these conditions is true:

- All of the fields have values (that is, are not Null).
- None of the fields has a value.

For example, if field1 = 5, field2 = "Mary", and field3 = null, **AllOrNone** returns False.

This function is useful, for example, where you have a set of page fields, and if any one of the fields contains a value, then all of the other fields are required also.

A blank character field, or a zero (0) numeric value in a required numeric field is considered a null value.

Related Functions

All Checks to see if a field contains a value, or if all the fields in a list of fields contain values. If any of the fields is Null, then **All** returns False.

None	Checks that a field or list of fields have no value. None is the opposite of All.
OnlyOne	Checks if exactly one field in the set has a value. Use this when the user must fill in only one of a set of mutually exclusive fields.
OnlyOneOrNone	Checks if no more than one field in the set has a value. Use this in cases when a set of fields is both optional and mutually exclusive; that is, if the user can put a value into one field in a set of fields, or leave them all empty.

Returns

Returns a Boolean value: True if all of the fields in fieldlist or none of the fields in fieldlist has a value, False otherwise.

Example

You could use **AllOrNone** as follows:

```
If Not AllOrNone(STREET1, CITY, STATE) Then
   WinMessage("Address should consist of at least Street (Line 1), City, State,⇒
 and Country.");
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, All , page 41, Chapter 1, PeopleCode Built-in Functions, None , page 335, Chapter 1, PeopleCode Built-in Functions, OnlyOne , page 344, Chapter 1, PeopleCode Built-in Functions, OnlyOneOrNone , page 345.

AllowEmplIdChg

Syntax

AllowEmplIdChg(*is_allowed*)

Description

By default, the Component Processor does not allow an user to make any changes to a record if a record contains an EMPLID key field and its value matches the value of the user's EMPLID. In some situations, though, such changes are warranted. For example, you would want employees to be able to change information about themselves when entering time sheet data.

AllowEmplIdChg enables the user to change records whose key matches the user's own EMPLID, or prevents the user from changing these records. The function takes a single Boolean parameter that when set to True allows the employee to update their own data. When the parameter is set to False, the employee is prevented from updating this data.

After permission is granted, it stays through the life of the *component*, not the page. After a user switches to another component, the default value (not being able to make changes) is reapplied.

Parameters

Parameter	Description
is_allowed	A Boolean value indicating whether the user is permitted to change the user's own data.

Returns

Optionally returns a Boolean value: True if the function executed successfully, False otherwise.

Example

```
If Substring (%Page, 1, 9) = Substring(PAGE.TimeSHEET_PNL_A, 1, 9) Then
   AllowEmplIdChg(true);
End-if;
```

Amortize

Syntax

Amortize(*intr*, *pb*, *pmt*, *pmtnbr*, *payintr*, *payprin*, *balance*)

Description

Amortize computes the amount of a loan payment applied towards interest (*payintr*), the amount of the payment applied towards principal (*payprin*), and the remaining balance *balance*, based on the principal balance (*pb*) at the beginning of the loan term, the amount of one payment *pmt*, the interest rate charged during one payment period (*intr*), and the payment number *pmtnbr*.

Parameters

Note that *payintr*, *payprin*, and *balance* are "outvars": you must pass variables in these parameters, which **Amortize** then fills with values. The remaining parameters are "invars" containing data the function needs to perform its calculation.

Parameter	Description
intr	Number indicating the percent of interest charged per payment period.
pb	Principal balance at the beginning of the loan term (generally speaking, the amount of the loan).
pmt	The amount of one loan payment.
pmtnbr	The number of the payment.
payintr	The amount of the payment paid toward interest.
payprin	The amount of the payment paid toward principal.
balance	The remaining balance after the payment.

Returns

None.

Example

Suppose you want to calculate the principal, interest, and remaining balance after the 24th payment on a loan of $15,000, at an interest rate of 1% per loan payment period, and a payment amount of $290.

```
&INTRST_RT=1;
&LOAN_AMT=15000;
&PYMNT_AMNT=290;
&PYMNT_NBR=24;
Amortize(&INTRST_RT, &LOAN_AMT, &PYMNT_AMNT, &PYMNT_NBR, &PYMNT_INTRST, ⇒
&PYMNT_PRIN, &BAL);
&RESULT = "Int=" | String(&PYMNT_INTRST) | " Prin=" | String(&PYMNT_PRIN) | "   ⇒
 Bal=" | String(&BAL);
```

This example sets &RESULT equal to "Int=114 Prin=176 Bal=11223.72".

Asin

Syntax

Asin(*value*)

Description

The **Asin** function calculates the arcsine of the given value, that is, the size of the angle whose sine is that value.

Parameters

Parameter	Description
value	Any real number between -1.00 and 1.00 inclusive, the range of valid sine values. If the input value is outside this range, an error message appears at runtime ("Decimal arithmetic error occurred. (2,110)"). Adjust your code to provide a valid input value.

Returns

A value in radians between -pi/2 and pi/2.

Example

The following example returns the size in radians of the angle whose sine is 0.5:

```
&MY_ANGLE = Asin(0.5);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Acos , page 29, Chapter 1, PeopleCode Built-in Functions, Atan , page 46, Chapter 1, PeopleCode Built-in Functions, Cos , page 91, Chapter 1, PeopleCode Built-in Functions, Cot , page 92, Chapter 1, PeopleCode Built-in Functions, Degrees , page 134, Chapter 1, PeopleCode Built-in Functions, Radians , page 356, Chapter 1, PeopleCode Built-in Functions, Sin , page 439, Chapter 1, PeopleCode Built-in Functions, Tan , page 456.

Atan

Syntax

Atan(*value*)

Description

The **Atan** function calculates the arctangent of the given value, that is, the size of the angle whose tangent is that value.

Parameters

Parameter	Description
value	Any real number.

Returns

A value in radians between -pi/2 and pi/2.

Example

The following example returns the size in radians of the angle whose tangent is 0.5:

```
&MY_ANGLE = Atan(0.5);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Acos , page 29, Chapter 1, PeopleCode Built-in Functions, Asin , page 45, Chapter 1, PeopleCode Built-in Functions, Cos , page 91, Chapter 1, PeopleCode Built-in Functions, Cot , page 92, Chapter 1, PeopleCode Built-in Functions, Degrees , page 134, Chapter 1, PeopleCode Built-in Functions, Radians , page 356, Chapter 1, PeopleCode Built-in Functions, Sin , page 439, Chapter 1, PeopleCode Built-in Functions, Tan , page 456.

BlackScholesCall

Syntax

BlackScholesCall(*Asset_Price*, *Strike_Price*, *Interest_Rate*, *Years*, *Volatility*)

Description

The **BlackScholesCall** function returns the value of a call against an equity underlying according to the Black-Scholes equations.

Parameters

Parameter	Description
Asset_Price	The asset price. This parameter takes a decimal value.
Strike_Price	The strike price. This parameter takes a decimal value.
Interest_Rate	The risk-free interest rate. This parameter takes a decimal value.
Years	The number of years to option expiration. This parameter takes a number value (decimal).
Volatility	The volatility of underlying. This parameter takes a decimal value.

Returns

A number representing the value of a call against an equity.

See Also

Chapter 1, PeopleCode Built-in Functions, BlackScholesPut , page 47.

BlackScholesPut

Syntax

```
BlackScholesPut(Asset_Price, Strike_Price, Interest_Rate, Years, Volatility)
```

Description

The **BlackScholesPut** function returns the value of a put against an equity underlying according to the Black-Scholes equations.

Parameters

Parameter	Description
Asset_Price	The asset price. This parameter takes a decimal value.
Strike_Price	The strike price. This parameter takes a decimal value.
Interest_Rate	The risk-free interest rate. This parameter takes a decimal value.
Years	The number of years to option expiration. This parameter takes a number (decimal) value.
Volatility	The volatility of underlying. This parameter takes a decimal value.

Returns

A number representing the value of a call against an equity.

See Also

Chapter 1, PeopleCode Built-in Functions, BlackScholesCall , page 46.

BootstrapYTMs

Syntax

BootstrapYTMs(*Date*, *MktInst*, *Accrual_Conv*)

Description

The **BootstrapYTMs** function creates a zero-arbitrage implied zero-coupon curve from a yield-to-maturity curve using the integrated discount factor method, based on the *Accrual_Conv*.

Parameters

Parameter	Description
Date	The trading date of the set of market issues. This parameter takes a date value.
MktInst	The market instrument properties. This parameter takes an array of array of number. The elements in the array specify the following:

Elements	Description
1	tenor in days
2	yield in percent
3	price per 100 par
4	coupon rate (zero for spot instruments)
5	frequency of coupon payments
6	unit of measure for coupon frequency, 0 for days, 1 for months, and 2 for years
7	coefficient **a** of a curve interpolating the dataset
8,9,10	coefficients **b**, **c**, and **d** of a curve interpolating the dataset

Returns

An array of array of number. The elements in the array have the same type as the elements in the array for the *MktInst* parameter.

See Also

Chapter 7, Array Class , page 633.

Break

Syntax

Break

Description

A **Break** statement takes no parameters and can appear within a loop or an **Evaluate** statement. It terminates execution of the loop or **Evaluate**, and resumes execution immediately after the end of the statement. If the loop or **Evaluate** is nested in another statement, only the innermost statement is terminated.

Example

In the following example, **Break** is used to terminate the **Evaluate** statement, while staying within the outermost **If** statement:

```
If CURRENCY_CD = PriorEffdt(CURRENCY_CD) Then
      Evaluate ACTION
      When = "PAY"
         If ANNUAL_RT = PriorEffdt(ANNUAL_RT) Then
            Warning MsgGet(1000, 27, "Pay Rate Change action is chosen and Pay⇒
 Rate has not been changed.");
         End-if;
         Break;
      When = "DEM"
         If ANNUAL_RT >= PriorEffdt(ANNUAL_RT) Then
            Warning MsgGet(1000, 29, "Demotion Action is chosen and Pay Rate has⇒
 not been decreased.");
         End-if;
        Break;
      When-other
      End-evaluate;
      WinMessage("This message appears after executing either of the BREAK⇒
 statements or after all WHEN statements are false");
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Evaluate , page 169, Chapter 1, PeopleCode Built-in Functions, Exit , page 176, Chapter 1, PeopleCode Built-in Functions, For , page 192, Chapter 1, PeopleCode Built-in Functions, While , page 502.

BulkDeleteField

Syntax

BulkDeleteField(*ProjectName*, **Field.***FieldName*, [*ExclProj*])

Description

Use the **BulkDeleteField** function to delete fields from records and pages, as well as the associated PeopleCode programs and modifying the SQL either on the record, or, if the record is a subrecord, on the parent records.

The field is removed from the page regardless of where the field exists on the page, whether on a grid or not.

If the field is in the SELECT clause of the SQL, the removal is straightforward. However, if the field is also used in a WHERE clause, or if the field is the only item in the SELECT clause, the record isn't modified and is instead inserted into a project called BLK_*FieldName*. The record should be examined and any additional changes made as necessary.

Deleting fields from records and pages does *not* remove the field definition itself and it does not remove the field from other areas, such as Projects, Crystal Reports, or message definitions.

In addition, this function does *not* delete the references to the field in the PeopleCode. You must manually remove the references to the deleted field. Use the Find In. . . tool to search for the field name you deleted.

Note. Because performing this operation changes records, you must subsequently rebuild the project (alter tables).

Using the Log File

Information about this operation is stored in a log field. The directory where the log file is placed depends on where the function is run from:

- If the function is run in two-tier, the log file is located here. This is also the default location if the system cannot find the other paths:

  PS_HOME/BulkOps.txt

- If the function is run from an application server, the log file is located here:

  PS_HOME/APPSERV/*Domain_Name*/LOGS/BulkOps.txt

- If the function is run from an Application Engine program, the log file is written to the process' output log directory, that is:

  PS_HOME/APPSERV/prcs/*Domain_Name*/log_output/*Process_Name_Instance*/BulkOps.txt

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on a currently loaded component. In general, changes aren't recognized until the component is reloaded.

Bulk operations are time consuming, therefore, referencing the log file to see the progress of an operation is recommended. These operations accommodate errors and continue processing, logging the overall progress of the operation.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Considerations Using the Exclusion Project

If you specify *ExclProj*, the following items that are both in *ProjectName* and *ExclProj* are *not* changed, that is, the field specified is *not* removed from these items:

- pages
- records

- associated SQL with records of type View
- any PeopleCode associated with those items.

Individual SQL or PeopleCode items are not ignored by themselves, only as associated with records or pages.

Parameters

Parameter	Description
ProjectName	The name of a project that is the source of records and pages to use.
FieldName	The name of the field to be deleted. This name must be prefixed with the reserved word **Field**.
ExclProj	The name of a project that has pages that should be ignored by this function.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Bulk operation completed successfully.
%MDA_Failure	Bulk operation did not complete successfully.

Example

```
&pjm = "MYPROJ";
&ret =  BulkDeleteField(&pjm, Field.OrgId, "EXCLPROJ");

If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "BulkDeleteField succeeded");
Else
MessageBox(0, "Metadata Fn Status", 0, 0, "BulkDeleteField failed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, BulkInsertField , page 51, Chapter 1, PeopleCode Built-in Functions, BulkModifyPageFieldOrder , page 54, *PeopleCode Developer's Guide*, Debugging Your Application, Find In . . ., *PeopleSoft Application Designer*, Appendix F: Performing Bulk Operations, *PeopleSoft Application Designer*, Building SQL Tables and Views.

BulkInsertField

Syntax

BulkInsertField(*ProjectName*, **Field.***FieldName*, *ModelName*, *ClonePCode* [, *ExclProj*])

Description

Use the **BulkInsertField** function to insert a source field into records and pages in a project if and only if the model field specified by *ModelName* exists on those records and pages.

where *Domain_Name* is the name of the application server domain.

Using the Log File

Information about this operation is stored in a log field. The directory where the log file is placed depends on where the function is run from:

- If the function is run in two-tier, the log file is located here. This is also the default location if the system cannot find the other paths:

  PS_HOME/BulkOps.txt

- If the function is run from an application server, the log file is located here:

  PS_HOME/APPSERV/*Domain_Name*/LOGS/BulkOps.txt

- If the function is run from an Application Engine program, the log file is written to the process' output log directory, that is:

  PS_HOME/APPSERV/prcs/*Domain_Name*/log_output/*Process_Name_Instance*/BulkOps.txt

Considerations Inserting Fields into Records

In records, the source field is assigned the same record field properties as the model field on each record, and is inserted directly after the model field.

If the model field has a prompt table, a prompt table is created for the source field using the name of the source field with TBL appended to it.

If the record is either a SQL View or Dynamic View type, the associated SQL is modified by having the SELECT clause expanded to include the new field.

If the record is a subrecord, the parent records of type SQL View or Dynamic View that contain this subrecord are updated.

If the SQL contains the model field in the WHERE clause, or the SQL is complex, the associated record is inserted into a project called BLK_*FieldName*. You should examine this record and make any necessary changes.

If the model field has PeopleCode associated with it on the record or in a component, and *ClonePCode* has been set to True, this PeopleCode is cloned to the new field, with all references to the model field changed to refer to the new field.

Note. Because using this function changes records that are used to build application tables, you must rebuild (alter) the specified project before these changes can be used.

Considerations Inserting Fields into Pages

If the model field is in a grid, the system inserts the new field into the grid next to the model field and assigns it the same page field properties.

If the model field is not in a grid, the system inserts the new field onto the page to the right of the model field (in the first open space) and assigns it the same page field properties. If the system detects a questionable field position, it inserts the page into a project called BLK_*FieldName*. The page will work as-is, however, the GUI layout may not be optimal, so you should examine these pages by hand.

The page field name property isn't cloned if it exists on the model field. Instead, the field name of the new field is used, since the page field name should be a unique identifier for page elements.

Note. If the project you specified only contained pages and not records, you do not need to rebuild the project after using this function. The changes take affect when the component containing the page is reloaded.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Bulk operations are time consuming, therefore, referencing the log file to see the progress of an operation is recommended. These operations accommodate errors and continue processing, logging the overall progress of the operation.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Considerations Using the Exclusion Project

If you specify *ExclProj*, the following items that are both in *ProjectName* and *ExclProj* are *not* changed, that is, the field specified is *not* inserted to these items:

- pages
- records
- associated SQL with records of type View
- any PeopleCode associated with those items.

Individual SQL or PeopleCode items are not ignored by themselves, only as associated with records or pages.

Parameters

Parameter	Description
ProjectName	The name of a project that is the source of records and pages to use.
FieldName	The name of the field to be inserted. This name must be prefixed with the reserved word **Field**.
ModelName	The name of a field on which to model the inserted field. Attributes are cloned from it for records and pages, PeopleCode is modified, and SQL inserted.
ClonePCode	Specify whether to clone the PeopleCode from the model to this field. This parameter takes a Boolean value: True, clone the PeopleCode programs, False, don't.
ExclProj	The name of a project that has pages that should be ignored by this function.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Bulk operation completed successfully.
%MDA_Failure	Bulk operation did not complete successfully.

Example

```
&pjm = "MYPROJ";

&ret =  BulkInsertField(&pjm, Field.OrgId, Field.DeptId, TRUE, "EXCLPROJ");

If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "BulkInsertField succeeded");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "BulkInsertField failed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, BulkModifyPageFieldOrder , page 54, Chapter 1, PeopleCode Built-in Functions, BulkDeleteField , page 49, *PeopleSoft Application Designer*, Appendix F: Performing Bulk Operations, *PeopleSoft Application Designer*, Building SQL Tables and Views.

BulkModifyPageFieldOrder

Syntax

BulkModifyPageFieldOrder({*ProjectName* | *PageList*}, *ColNames*, *RequireAll*, [*ColWidths*])

Description

Use the **BulkModifyPageFieldOrder** function to reorder the grid columns as specified by *ColNames*. If *ColWidths* is specified, the columns are also resized. This can also be used to modify a single columns width.

The reordering algorithm “bunches” these fields together at the first instance of any of these fields in a target page grid, and forces the remaining fields into the order specified.

This function only reorders fields inside a grid.

If the fields occur twice or more in a grid, from two or more records, such as work records, the fields are bunched together in record groupings before being sorted into the order specified. For example, the two records ABS_HIST and PERSONAL_HISTORY both contain the fields DURATION_DAYS and DURATION_HOURS. The following is an example of how the records are fields would be bunched together first:

- ABS_HIST, DURATION_DAYS
- ABS_HIST, DURATION_HOURS
- PERSONAL_HISTORY, DURATION_DAYS
- PERSONAL_HISTORY, DURATION_HOURS

Note. These changes take affect after components are reloaded.

Using the Log File

Information about this operation is stored in a log field. The directory where the log file is placed depends on where the function is run from:

- If the function is run in two-tier, the log file is located here. This is also the default location if the system cannot find the other paths:

  PS_HOME/BulkOps.txt

- If the function is run from an application server, the log file is located here:

  PS_HOME/APPSERV/*Domain_Name*/LOGS/BulkOps.txt

- If the function is run from an Application Engine program, the log file is written to the process' output log directory, that is:

  PS_HOME/APPSERV/prcs/*Domain_Name*/log_output/*Process_Name_Instance*/BulkOps.txt

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Bulk operations are time consuming, therefore, referencing the log file to see the progress of an operation is recommended. These operations accommodate errors and continue processing, logging the overall progress of the operation.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
ProjectName | *PageList*	Specify either the name of a project that is the source of pages to use or an array of page names.
ColNames	Specify an array of string that indicate which fields and the desired order of those fields.
RequireAll	Specify whether all the fields in *ColNames* must be present before changes are made or not. This parameter takes a Boolean value: True, all fields must be present.
ColWidths	Specify an array of number that gives the pixel widths of the grid columns. Use a -1 to indicate that the width of a column shouldn't change.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Bulk operation completed successfully.
%MDA_Failure	Bulk operation did not complete successfully.

Example

```
Local Array of String &ColOrder;
Local Array of Number &ColWidth;
Local String &pjm, &ret;

&pjm = "MYPROJ";
&ColWidth = CreateArray(50, 100, -1);
&ColOrder = CreateArray("DEPTID", "ORGID", "PROJECT");

&ret =  BulkModifyPageFieldOrder(&pjm, &ColOrder, TRUE, &ColWidth);

If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "BulkModifyPageFieldOrder⇒
 succeeded");
Else
      MessageBox(0, "Metadata Fn Status", 0, 0, "BulkModifyPageFieldOrder⇒
 failed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, BulkInsertField , page 51, Chapter 1, PeopleCode Built-in Functions, BulkDeleteField , page 49, *PeopleSoft Application Designer*, Appendix F: Performing Bulk Operations, *PeopleSoft Application Designer*, Building SQL Tables and Views.

BulkUpdateIndexes

Syntax

BulkUpdateIndexes([*StringFieldArray*])

Description

Use **BulkUpdateIndexes** to update indexes (PSINDEXDEFN table) for records that contain a field whose NotUsed setting has changed.

A field whose NotUsed flag has been set to True does *not* show up in indexes. The only way to modify a field's NotUsed setting is through an API call such as in the following example:

```
SetDBFieldNotUsed(FIELD.OrgId, True);
```

The indexes of records that contain this field need to be updated to reflect the new settings.

Information about this operation can be logged by turning on PeopleCode tracing of internal functions (value 256.)

Considerations Using this Function

Do *not* invoke this function from runtime pages, as it modifies all records, including the records used to support the page it is invoked from. This function should be invoked from a PeopleSoft Application Engine program.

Note. If you do call this function from a page the operation completes successfully, but the page returns an error message. Switching to a new component clears up this error, however, any changes not saved to the database are lost.

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Bulk operations are time consuming, therefore, referencing the log file to see the progress of an operation is recommended. These operations accommodate errors and continue processing, logging the overall progress of the operation.

Calling this function without any parameter rebuilds the indexes for *all* records, an operation that may take hours. By indicating a list of fields whose NotUsed flag has changed, only the affected records have their indexes updated, reducing the time required to run this function.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
StringFieldArray	Specify an array of field names (as strings), such as DEPTID, representing the fields whose NotUsed flag has been modified. Only the records containing these fields are updated. If you don't specify a value for this parameter, the indexes for *all* records are rebuilt.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Bulk operation completed successfully.
%MDA_Failure	Bulk operation did not complete successfully.

Example

The following example uses the function without the optional array of field names:

```
&ret =  BulkUpdateIndexes();
If (&ret = %MDA_Success) Then
MessageBox(0, "MetaData Fn Status", 0, 0, "BulkUpdateIndexes succeeded");
Else
MessageBox(0, "MetaData Fn Status", 0, 0, "BulkUpdateIndexes failed");
End-If;
```

The following example uses the function with an array of two field names passed to it:

```
&ret =  BulkUpdateIndexes(CreateArray("DEPTID","PROJECT"));
If (&ret = %MDA_success) Then
MessageBox(0, "MetaData Fn Status", 0, 0, "BulkUpdateIndexes succeeded");
Else
MessageBox(0, "MetaData Fn Status", 0, 0, "BulkUpdateIndexes failed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, BulkDeleteField , page 49, Chapter 1, PeopleCode Built-in Functions, BulkInsertField , page 51, Chapter 1, PeopleCode Built-in Functions, BulkModifyPageFieldOrder , page 54.

CallAppEngine

Syntax

CallAppEngine(*applid* [*, statereclist, processinstance*]);

Where *statereclist* is list of record objects in the form:
&staterecord1 [*, &staterecord2*] . . .

There can be only as many record objects in *statereclist* as there are state records for the Application Engine program. Additional record objects are ignored.

Description

CallAppEngine starts the PeopleSoft Application Engine program named *applid*. This is how to start your Application Engine programs synchronously from a page. (Prior to PeopleTools 8, you could do only this using **RemoteCall**.) Normally, you won't run PeopleSoft Application Engine programs from PeopleCode in this manner. Rather, the bulk of your Application Engine execution will be run using the Process Scheduler, and the exception would be done using **CallAppEngine**.

The *staterecord* can be the hard-coded name of a record, but generally you use a record object to pass in values to seed particular state fields. The record name must match the state record name exactly.

The *processinstance* allows you to specify the process instance used by the Application Engine runtime. In your PeopleCode program this parameter must be declared of type integer since that is the only way the runtime can tell whether the last parameter is to be interpreted as a process instance. For more details see the Application Engine documentation.

Note. If you use this function, you shouldn't use the %TruncateTable or %Execute meta-SQL statement in any of your Application Engine steps. This is because on some platforms an implicit commit occurs after these statements, and all online processing should be done as a single logical unit of work.

After you use **CallAppEngine**, you may want to refresh your page. The **Refresh** method, on a rowset object, reloads the rowset (scroll) using the current page keys. This causes the page to be redrawn. `GetLevel0().Refresh()` refreshes the entire page. If you want only a particular scroll to be redrawn, you can refresh just that part.

See *PeopleSoft Application Engine*, Using Meta-SQL and PeopleCode, Using PeopleCode in Programs.

Note. If you supply a non-zero process instance, all message logging is done under the process instance. You *must* build your own PIA page to access or delete the messages, since there is no Process Monitor entry for the process instance you used.

PeopleCode Event Considerations

You must include the CallAppEngine PeopleCode function within events that allow database updates because generally, if you're calling Application Engine, you're intending to perform database updates. This includes the following PeopleCode events:

- SavePreChange (Page)
- SavePostChange (Page)
- Workflow
- Message Subscription
- FieldChange

If **CallAppEngine** results in a failure, all database updates is rolled back. All information the user entered into the component is lost, as if the user pressed ESC.

Application Engine Considerations

You can use the **CallAppEngine** function in a PeopleSoft Application Engine program, either directly (in a PeopleCode action) or indirectly (using a Component Interface). This functionality must be used carefully, and you should only do this once you have a clear understanding of the following rules and restrictions.

- Dedicated cursors are *not* supported inside a "nested application engine instance" (meaning an application engine program invoked using CallAppEngine from within another application engine program). If a nested application engine instance has any SQL actions with ReUse set to Yes or Bulk Insert, those settings are ignored.
- As in any other type of PeopleCode event, no commits are performed within the called application engine program. This is an important consideration. If a batch application engine program called another program using CallAppEngine, and that child program updated many rows of data, the unit-of-work might become too large, resulting in contention with other processes. A batch application engine program should invoke such child programs using a Call Section action, not CallAppEngine.
- Temp tables are *not* shared between a batch application engine program and child program invoked using CallAppEngine. Instead, the child program is assigned an "online" temporary table instance, which is used for all temp tables in that program. In addition, if that child program invokes another program using CallAppEngine, that grandchild shares the online temp instance with the caller. In other words, only one online temp instance is allocated to a process at any one time, no matter how many nested CallAppEngine's there might be.
- The lock on an online temp instance persists until the next commit. If the processing time of the called program is significant (greater than a few seconds), this would be unacceptable. As a general rule, application engine programs that make use of temp tables and have a significant processing time should be called from other application engine programs using a Call Section action, not CallAppEngine.

Save Events Considerations

To execute the Application Engine program based on an end user Save, use the CallAppEngine function within a Save event. When you use CallAppEngine, you should keep the following items in mind:

- No commits occur during the entire program run.
- During SavePreChange, any modified rows in the page have not been written to the database.
- During SavePostChange, the modified rows have been written to the database. The Page Process issues one commit at the end of the Save cycle.

FieldChange Considerations

If you don't want the CallAppEngine call to depend on a Save event, you can also initiate CallAppEngine from a FieldChange event. When having a FieldChange event initiate CallAppEngine, keep the following items in mind:

- No commits occur within the program called by CallAppEngine. The called program remains a synchronous execution in the same unit of work.
- The normal FieldChange commit occurs, which frees any locks that the Application Engine program might have acquired.
- Do not include a DoSave function in the same FieldChange event. Not only is this not allowed, but it also indicates that you should be including the CallAppEngine within a Save event.
- You can use the DoSaveNow function in the same FieldChange event, but it must be called prior to the first CallAppEngine function, but not afterward.

Parameters

Parameter	Description
applid	Specify the name of the Application Engine program you want to start.
statereclist	Specify an optional record object that provides initial values for a state record.
processinstance	Specify the process instance used by the Application Engine runtime.

Returns

None.

Example

The following calls the Application Engine program named MYAPPID, and passes initialization values.

```
&REC = CreateRecord(RECORD.INIT_VALUES);
&REC.FIELD1.Value = "XYZ";
   /* set the initial value for INIT_VALUES.FIELD1 */
CallAppEngine("MYAPPID", &REC);
```

See Also

Chapter 1, PeopleCode Built-in Functions, DoSaveNow , page 155, Chapter 5, AESection Class , page 591, Chapter 27, Rowset Class , Refresh, page 1495, *PeopleSoft Application Engine*, Using Meta-SQL and PeopleCode, Using PeopleCode in Programs, *PeopleSoft Application Engine*, Creating Application Engine Programs, Call Section Actions.

CancelPubHeaderXmlDoc

Syntax

CancelPubHeaderXmlDoc(*PubID, PubNode, ChannelName, VersionName*)

Description

The message header, also known as the message instance, is the published message *before* any transformations were performed. The **CancelPubXmlDoc** method allows you to programmatically cancel the message header of a publication contract, much the same as you can do in the message monitor.

The function is only available when the message has one of the following statuses:

- Error
- New
- Retry
- Timeout
- Edited

Parameters

Parameter	Description
PubID	Specify the PubID of the message.
PubNode	Specify the Pub Node Name of the message.
ChannelName	Specify the channel name of the message.
VersionName	Specify the version name of the message.

Returns

A Boolean value: True if the function completed successfully, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, ReSubmitPubHeaderXmlDoc , page 374.

CancelPubXmlDoc

Syntax

CancelPubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName, SubNode*)

Description

The **CancelPubXmlDoc** method allows you to programmatically cancel a message publication contract, much the same as you can do in the message monitor. This is the publication contract that exists *after* any transformations have been preformed.

The function is only available when the message has one of the following statuses:

- Error
- New
- Retry
- Timeout
- Edited

Parameters

Parameter	Description
PubID	Specify the PubID of the message.
PubNode	Specify the Pub Node Name of the message.
ChannelName	Specify the channel name of the message.
VersionName	Specify the version name of the message.
MessageName	Specify the name of the message.
SubNode	Specify the subnode of the message.

Returns

A Boolean value: True if the function completed successfully, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, ReSubmitPubXmlDoc , page 375.

CancelSubXmlDoc

Syntax

CancelSubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName, SubscriptionName*)

Description

The **CancelSubXmlDoc** function enables you to programmatically cancel a message subscription contract, much the same as you can do in the message monitor.

The function is only available when the message has one of the following statuses:

- Error
- New
- Retry

- Timeout
- Edited

Parameters

Parameter	Description
PubID	Specify the PubID as a string.
PubNode	Specify the Pub Node name as a string.
ChannelName	Specify the Channel name as a string.
VersionName	Specify the version name as a string.
MessageName	Specify the message name as a string.
SubscriptionName	Specify the subscription name as a string.

Returns

A Boolean value: True if function completed successfully, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, ReSubmitSubXmlDoc , page 376.

ChangeEmailAddress

Syntax

```
ChangeEmailAddress(Type, Address)
```

Description

Use the **ChangeEmailAddress** function to change the type of an email address that you've already added for the current user. You can only have *one* email address of a specific type for a user. If you try to use a type that already has an email address associated with it, you receive an error.

Parameters

Parameter	Description
Type	Specify the type that you want to change the email address to. This parameter takes a string value. The valid values are:

Value	Description
BB	Blackberry email address
BUS	Business email address
HOME	Home email address
OTH	Other email address
WORK	Work email address

Parameter	Description
Address	Specify the email address that you want to add as a string.

Returns

None.

See Also

Chapter 1, PeopleCode Built-in Functions, AddEmailAddress , page 35, Chapter 1, PeopleCode Built-in Functions, DeleteEmailAddress , page 137, Chapter 1, PeopleCode Built-in Functions, MarkPrimaryEmailAddress , page 317.

Char

Syntax

Char(*n*)

Description

Char converts a numeric value to the corresponding character in the current character code set.

Returns

Returns a String representing the ANSI character corresponding to the number *n*.

Example

The examples set &LETTER to "P", then "S".

```
&LETTER = Char(80);
&LETTER = Char(83);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Code , page 69, Chapter 1, PeopleCode Built-in Functions, String , page 451, Chapter 2, Meta-SQL , %Substring , page 550.

CharType

Syntax

CharType(*source_str, char_code*)

Description

CharType determines whether the first character in *source_str* is of type *char_code* **.** The *char_code* is a number value representing a character set (see the following Parameters section for details).

Japanese character sets can be recognized only if a Japanese operating system is running. If the function is unable to determine whether the first character in *source_str* belongs to the specified character set, the function returns a value of UNKNOWN (-1). This occurs if the application is running on a non-Japanese version of Windows or, if on a UNIX application server, the CharacterSet parameter in the PSADMIN table isn't set to SJIS.

See Also

PeopleSoft Application Engine, Creating Application Engine Programs, Call Section Actions.

Parameters

Parameter	Description
source_str	A String, the first character of which will be tested.
char_code	A Number representing the character type to be tested for.

The following table shows valid values valid values for *char_code*. You can specify either a Character Code or a Constant:

Numeric Value	Constant	Character Set
0	%CharType_AlphaNumeric	Alphanumeric (7-bit ASCII codes; A-Z, a-z, 1-9, punctuation)
1	%CharType_ExtendedLatin1	Extended Latin-1 characters (ISO8859-1 accents for Spanish, French, and so on.)
2	%CharType_HankakuKatakana	Hankaku Katakana (single-byte Japanese Katakana)
3	%CharType_ZenkakuKatakana	Zenkaku Katakana (double-byte Japanese Katakana)
4	%CharType_Hiragana	Hiragana (Japanese)
5	%CharType_Kanji	Kanji (Japanese)
6	%CharType_DBAlphaNumeric	Double-byte Alphanumeric (Japanese)
7,8,9		Reserved for future use
10	%CharType_JapanesePunctuation	Japanese punctuation

Returns

CharType returns one of the following Number values. You can check for the constant values instead of the numeric values if you prefer:

Numeric Value	Constant Value	Description
1	%CharType_Matched	Character is of type *char_code*.
0	%CharType_NotMatched	Character is not of type *char_code*.
-1	%CharType_Unknown	UNKNOWN: unable to determine whether character is of set *char_code*. This occurs if the application is running on a non-Japanese version of Windows or on a UNIX application server when the CharacterSet parameter of the PSADMIN table is not set to SJIS.

Example

This example tests to see if a character is Kanji:

```
&ISKANJI = CharType(&STRTOTEST, 5);
If &ISKANJI = 1 Then
   WinMessage("Character type is Kanji");
Else
   If &ISKANJI = 0 Then
      WinMessage("Character type is Kanji");
   Else
      WinMessage("Character type is UNKNOWN");
   End-If
End-If
```

See Also

Chapter 1, PeopleCode Built-in Functions, ContainsCharType , page 75, Chapter 1, PeopleCode Built-in Functions, ContainsOnlyCharType , page 77, Chapter 1, PeopleCode Built-in Functions, ConvertChar , page 79, *PeopleTools Global Technology*, Character Sets and Language Input/Output.

ChDir

Syntax

ChDir(*path*)

Description

ChDir, like the DOS ChDir command, changes the current directory on a drive. The drive and the directory are both specified in a *path* string.

Note. This function has been deprecated.

ChDrive

Syntax

ChDrive(*str_dr*)

Description

ChDrive changes the current disk drive to the drive specified by *str_dr*, which is a string consisting of a valid drive letter followed by a colon, for example "C:".

Note. This function has been deprecated.

CheckMenuItem

Syntax

```
CheckMenuItem(BARNAME.menubar_name, ITEMNAME.menuitem_name)
```

Description

CheckMenuItem changes the menu state by placing a check mark beside the menu item.

Note. This function has been deprecated.

Clean

Syntax

```
Clean(string)
```

Description

Clean removes all non-printable characters from a text *string* and returns the result as a String value. It is intended for use on text imported from other applications that contains characters that may not print correctly under Windows. Frequently, low-level characters appear at the beginning and end of each line of imported data, and they cannot be printed.

Returns

Returns a String value purged of non-printable characters.

Example

Because **Char**(7) is a non-printable character, the following **Clean** function returns a null string:

```
&CLEANSTR = Clean(Char(7));
```

See Also

Chapter 1, PeopleCode Built-in Functions, Char , page 64, Chapter 1, PeopleCode Built-in Functions, String , page 451, Chapter 2, Meta-SQL , %Substring , page 550.

ClearKeyList

Syntax

```
ClearKeyList( )
```

Description

ClearKeyList clears the current key list. This function is useful for programmatically setting up keys before transferring to another component.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Example

The following example sets up a key list and then transfers the user to a page named PAGE_2.

```
ClearKeyList( );
AddKeyListItem(OPRID, OPRID);
AddKeyListItem(REQUEST_ID, REQUEST_ID);
SetNextPage("PAGE_2");
TransferPage( );
```

See Also

Chapter 1, PeopleCode Built-in Functions, AddKeyListItem , page 36.

ClearSearchDefault

Syntax

ClearSearchDefault([*recordname*.]*fieldname*)

Description

ClearSearchDefault disables default processing for the specified field, reversing the effects of a previous call to **SetSearchDefault**.

Note. This function remains for backward compatibility only. Use the **SearchDefault** Field class property instead.

If search default processing is cleared for a record field, the default value specified in the record field properties for that field will not be assigned when the field appears in a search dialog box. This function is effective only when used in SearchInit PeopleCode.

See Also

Chapter 12, Field Class , SearchDefault, page 865, *PeopleCode Developer's Guide*, Data Buffer Access.

Parameters

Parameter	Description
[*recordname* .]*fieldname*	The name of the target field, which is a search key or alternate search key that is about to appear in a search dialog box. The *recordname* prefix is required if the call to **ClearSearchDefault** is in a record definition other than *recordname*.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

See Also

Chapter 1, PeopleCode Built-in Functions, ClearSearchEdit , page 69, Chapter 1, PeopleCode Built-in Functions, SetSearchDefault , page 428, Chapter 1, PeopleCode Built-in Functions, SetSearchEdit , page 430, Chapter 1, PeopleCode Built-in Functions, SetSearchDialogBehavior , page 429, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Search Processing in Update Modes.

ClearSearchEdit

Syntax

ClearSearchEdit([*recordname*.]*fieldname*)

Description

ClearSearchEdit reverses the effects of a previous call to **SetSearchEdit**. If ClearSearchEdit is called for a specific field, the edits specified in the record field properties will not be applied to the field when it occurs in a search dialog.

Note. This function remains for backward compatibility only. Use the SearchEdit Field class property instead.

See Also

Chapter 12, Field Class , SearchEdit, page 865, *PeopleCode Developer's Guide*, Data Buffer Access.

Parameters

Parameter	Description
[*recordname* **.**]*fieldname*	The name of the target field, which is a search key or alternate search key about to appear in a search dialog box. The *recordname* prefix is required if the call to ClearSearchEdit is in a record definition other than *recordname*.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

See Also

Chapter 1, PeopleCode Built-in Functions, SetSearchEdit , page 430, Chapter 1, PeopleCode Built-in Functions, SetSearchDefault , page 428, Chapter 1, PeopleCode Built-in Functions, ClearSearchDefault , page 68, Chapter 1, PeopleCode Built-in Functions, SetSearchDialogBehavior , page 429, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Search Processing in Update Modes.

Code

Syntax

Code(*str*)

Description

Code returns a numeric code for the first character in a text *str*. (Normally you would pass this function a single character.) The return code depends on the character set in use by the operating system. If the first character of *str* consists of a lead byte and trailing byte (in a double-byte character set), Code returns the numeric equivalent of both bytes.

Returns

Returns a Number value equal to the character code for the first character in *str.*

See Also

Chapter 1, PeopleCode Built-in Functions, Char , page 64, Chapter 1, PeopleCode Built-in Functions, String , page 451, Chapter 1, PeopleCode Built-in Functions, Codeb , page 70, Chapter 2, Meta-SQL , %Substring , page 550.

Codeb

Syntax

```
Code(str)
```

Description

Codeb is a byte-oriented variant of **Code**, which determines the value of the first byte in *str*.

Returns

Returns a Number value equal to the first byte in *str*.

See Also

Chapter 1, PeopleCode Built-in Functions, Char , page 64, Chapter 1, PeopleCode Built-in Functions, String , page 451, Chapter 1, PeopleCode Built-in Functions, Code , page 69, Chapter 2, Meta-SQL , %Substring , page 550.

CollectGarbage

Syntax

```
CollectGarbage()
```

Description

Use the **CollectGarbage** function to remove any unreachable application objects created by the Application Classes.

Sometimes there may be unrecovered application objects that are can no longer be referenced from PeopleCode, but which have not been reclaimed and so are still taking up computer memory. Generally this situation arises only if you have application objects that form into loops of references.

This function is automatically invoked by the application server as part of its end-of-service processing, so generally you don't need to call it for online applications. However, in Application Engine (batch), it is possible that a long-running batch job could grow in memory usage over time as these unreferencable Application Objects accumulate. The solution to such a problem is to call CollectGarbage periodically to reclaim these objects.

Parameters

None.

Returns

None.

See Also

Chapter 6, Application Classes , page 601.

CommitWork

Syntax

```
CommitWork()
```

Description

The **CommitWork** function commits pending changes (inserts, updates, and deletes) to the database.

Considerations for Using CommitWork

The following are the considerations for using CommitWork.

- This function is available in Application Engine PeopleCode, the FieldChange and SavePreChange events, and Message Subscription PeopleCode. If you use it anywhere else, you'll receive a runtime error.
- When used with an Application Engine program, this function only applies to those Application Engine programs that run in batch (not online). If the program is invoked using **CallAppEngine**, the CommitWork call is ignored. The same is true for commit settings at the section or step level.
- This function can only be used in an Application Engine program that has restart disabled. If you try to use this function in a program that doesn't have restart disabled, you'll receive a runtime error.

See Chapter 1, PeopleCode Built-in Functions, CallAppEngine , page 58.

FieldChange and SavePreChange Considerations

The following are the FieldChange and SavePreChange considerations:

- All updates done in FieldChange (including those using CallAppEngine) should be considered a *single database transaction*. This is a fundamental change: previously, a single transaction was represented by a page or a component.
- A consequence of this is that a message requiring a reply, or any other think-time action, causes a *fatal* error if located in FieldChange *after* a database update that has *not* been committed to the database using the CommitWork function. So it is possible for an application to update the database in FieldChange, then do a think-time action, by preceding the think-time action with a call to CommitWork.
- CommitWork commits the updates and closes the database transaction (that is, the unit of work). The consequence of using CommitWork is that because it closes the database transaction, any subsequent rollback calls will *not* rollback the committed updates.
- Just as any database updates in FieldChange required careful application design to ensure that the transaction model is appropriate, so too does the use of CommitWork.
- When using CommitWork in the Component Processor environment (as opposed to using it in an Application Engine program) CommitWork produces an error if there are any open cursors, such as any open PeopleCode SQL objects.

See *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, FieldChange Event.

Application Engine Considerations

The CommitWork function is useful only when you are doing row-at-a-time SQL processing in a single PeopleCode program, and you must commit without exiting the program. In a typical Application Engine program, SQL commands are split between multiple Application Engine actions that fetch, insert, update, or delete application data. Therefore, you would use the section or step level commit settings to manage the commits. This is the recommended approach.

However, with some types of Application Engine programs that are PeopleCode intensive, it can be difficult to exit the PeopleCode in order to perform a commit. This is the only time when the CommitWork function should be used.

See *PeopleSoft Application Engine*, Managing Application Engine Programs, Restarting Application Engine.

Restart Considerations

Disabling restart on a particular program means that the application *itself* is intrinsically self-restartable: it can be re-run from the start after an abend, and it performs any initialization, cleanup, and filtering of input data to ensure that everything gets processed once and only once, and that upon successful completion, the database is in the same state it would have been if no abend occurred.

Set-based applications should always use Application Engine's restart. Only row-by-row applications that have restart built into them can benefit from disabling Application Engine's restart.

Consider the following points to managing restarts in a self-restarting program:

- Locking input transactions (optional).

  If the input data can change, and if it's important not to pick up new data during a restart, there should be logic to lock transactions at the start of the initial run (such as updating rows with current Process Instance). The program should first check whether any rows have the current Process Instance (that is, is the process being restarted from the top after an abend?). If no rows found, do the update.

  In some cases it is acceptable for a restarted process to pick up new rows, so that locking is not necessary. It depends on your application.

  Also, if you don't lock transactions, you must provide some other way to manage concurrent processing of the same program. You don't want two simultaneous runs of the same program to use the same data, so you must have some strategy for dividing up the data such that there is no overlap.

- Filtering input transactions (required).

  After an input transaction is processed, the row should be updated accordingly (that is, setting a "processed" flag). The SELECT statement that drives the main processing loop should include a WHERE condition to filter out rows that have already been processed.

Returns

A Boolean value, True if data was successfully committed, False otherwise.

Example

The following example fetches rows and processes them one at a time, committing every 100 iterations. Because restart is disabled, you must have a marker indicating which rows have been processed, and use it in a conditional clause that filters out those rows.

```
Local SQL &SQL;
```

```
Local Record &REC;
Local Number &COUNT;

&REC = CreateRecord(RECORD.TRANS_TBL);
&SQL = CreateSQL("%SelectAll(:1) WHERE PROCESSED <> 'Y'");
&COUNT = 0;

&SQL.Execute(&REC);
While &SQL.Fetch(&REC)
   If (&COUNT > 99) Then
      &COUNT = 0;
      CommitWork();   /* commit work once per 100 iterations */
   End-if;
   &COUNT = &COUNT + 1;
   /* do processing */
   ...

   /* update transaction as "processed" */
   &REC.PROCESSED.Value = 'Y';
   &REC.Update();
End-While;
```

See Also

PeopleSoft Application Engine, Using Meta-SQL and PeopleCode, Using PeopleCode in Programs.

CompareLikeFields

Syntax

CompareLikeFields(*from*, *to*)

where *from* and *to* are constructions that reference rows of data on specific source and target records in the component buffer; each have the following syntax:

level, *scrollpath*, *target_row*

and where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD.***target_recname*

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

CompareLikeFields compares fields in a row on a specified source record to similarly named fields on a specified target record.

Note. This function remains for backward compatibility only. Use the **CompareFields** record class method instead.

If all of the like-named fields have the same data value, CompareLikeFields returns True; otherwise it returns False.

See Also

Chapter 25, Record Class , CompareFields, page 1443, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
from	A placeholder for a construction (*level*, *scrollpath*, *target_row*) that references the first row in the comparison.
to	A placeholder for a construction (*level*, *scrollpath*, *target_row*) that references the second row in the comparison.
level	Specifies the scroll level for the target level scroll.
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	Specifies the row number of each target row on its scroll level.

Returns

Returns a Boolean value indicating whether all of the like-named fields in the two records have the same data value.

Example

The following example compares the like-named fields in two rows on levels 1 (&L1_ROW) and 2 (&L2_ROW) and returns True if all of the like-named fields in the two rows have the same value.

```
&L1_ROW = 1;
&L2_ROW = 1;
if CompareLikeFields(1, RECORD.BUS_EXPENSE_PER, &L1_ROW, 2, ⇒
RECORD.BUS_EXPENSE_PER, 1, RECORD.BUS_EXPENSE_DTL, &L2_ROW) then
   WinMessage("The fields match.");
end-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CopyFields , page 89.

Component

Syntax

Component *data_type &var_name*

Description

The **Component** statement enables you to declare PeopleCode Component variables. A Component variable, after being declared in any PeopleCode program, remains in scope throughout the life of the component.

The variable must be declared with the Component statement in *every* PeopleCode program in which it is used.

Declarations appear at the beginning of the program, intermixed with function declarations.

Note. Because a function can be called from anywhere, you cannot declare any variables within a function. You receive a design time error if you try.

The system automatically initializes temporary variables. Declared variables always have values appropriate to their declared type. Undeclared variables are initialized as null strings.

Not all PeopleCode data types can be declared as Component.

Parameters

Parameter	Description
data_type	Specify a PeopleCode data type.
&*var_name*	A legal variable name.

Example

```
Component string &PG_FIRST;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Local , page 315, Chapter 1, PeopleCode Built-in Functions, Global , page 281, *PeopleCode Developer's Guide*, PeopleCode Language, Data Types.

ComponentChanged

Syntax

ComponentChanged()

Description

ComponentChanged determines whether a component has changed since the last save, whether by the user or by PeopleCode.

Returns

Returns a Boolean value: True if the component has changed.

Example

```
If ComponentChanged() Then
   /* do some stuff */
End-if;
```

ContainsCharType

Syntax

ContainsCharType(*source_str*, *char_code*)

Description

ContainsCharType determines if any of the characters in *source_str* are of type *char_code.* The *char_code* is a number value representing a character set (see the following Parameters section for details).

Japanese character sets can be recognized only if a Japanese operating system is running. UNIX application servers support only alphanumeric and extended Latin character sets. If the function is unable to determine whether *source_str* contains any characters belonging to the specified character set, the function returns a value of UNKNOWN (-1). This occurs if the application is running on a non-Japanese version of Windows or, if on a UNIX application server, the CharacterSet parameter in the PSADMIN table isn't set to SJIS.

Parameters

Parameter	Description
source_str	String to be examined.
char_code	A Number value representing the character set to be tested for. The following table shows valid values. You can specify either a character code numeric value or a constant:

Numeric Value	Constant	Character Set
0	%CharType_AlphaNumeric	Alphanumeric (7-bit ASCII codes; A-Z, a-z, 1-9, punctuation)
1	%CharType_ExtendedLatin1	Extended Latin-1 characters (ISO8859-1 accents for Spanish, French, etc.)
2	%CharType_HankakuKatakana	Hankaku Katakana (single-byte Japanese Katakana)
3	%CharType_ZenkakuKatakana	Zenkaku Katakana (double-byte Japanese Katakana)
4	%CharType_Hiragana	Hiragana (Japanese)
5	%CharType_Kanji	Kanji (Japanese)
6	%CharType_DBAlphaNumeric	Double-byte Alphanumeric (Japanese)
7,8,9		Reserved for future use
10	%CharType_JapanesePunctuation	Japanese punctuation

Returns

ContainsCharType returns one of the following Number values. You can check for the constant instead of the numeric value if you prefer:

Numeric Value	Constant Value	Description
1	%CharType_Matched	String contains at least one character of set *char_code*.
0	%CharType_NotMatched	String contains no characters of set *char_code*.
-1	%CharType_Unknown	UNKNOWN: unable to determine whether one or more characters in string is of set *char_code*. This occurs if the application is running on a non-Japanese version of Windows or on a UNIX application server when the CharacterSet parameter of the PSADMIN table is not set to SJIS.

Example

This example tests to see if the string contains any Kanji:

```
&ANYKANJI = ContainsCharType(&STRTOTEST, 5);
If &ANYKANJI = 1 Then
   WinMessage("There are Kanji characters");
Else
   If &ANYKANJI = 0 Then
      WinMessage("There are no Kanji characters");
   Else
      WinMessage("UNKNOWN");
   End-If
End-If
```

See Also

Chapter 1, PeopleCode Built-in Functions, ContainsCharType , page 75, Chapter 1, PeopleCode Built-in Functions, ContainsOnlyCharType , page 77, Chapter 1, PeopleCode Built-in Functions, ConvertChar , page 79, *PeopleTools Global Technology*, Character Sets and Language Input/Output.

ContainsOnlyCharType

Syntax

ContainsOnlyCharType(*source_str, char_code_list*)

Where *char_code_list* is a list of character set codes in the form:
char_code_1 [, *char_code_2*]. . .

Description

ContainsOnlyCharType determines whether every character in *source_str* belongs to one or more of the character sets in *char_code_list*. See the following Parameters section for a list of valid character code values.

Japanese character sets can be recognized only if a Japanese operating system is running. UNIX application servers support only alphanumeric and extended Latin character sets. If the function is unable to recognize one or more characters in *source_str*, the function returns a value of UNKNOWN (-1). This occurs if the application is running on a non-Japanese version of Windows or, if on a UNIX application server, the CharacterSet parameter in the PSADMIN table isn't set to SJIS.

Parameters

Parameter	Description
Source_str	String to be examined.
char_code_list	A comma-separated list of character set codes.
char_code_n	Either a Number value identifying a character set, or a constant. The following table shows valid values. You can specify either a character code numeric value or a constant:

Numeric Value	Constant	Character Set
0	%CharType_AlphaNumeric	Alphanumeric (7-bit ASCII codes; A-Z, a-z, 1-9, punctuation)
1	%CharType_ExtendedLatin1	Extended Latin-1 characters (ISO8859-1 accents for Spanish, French, etc.)
2	%CharType_HankakuKatakana	Hankaku Katakana (single-byte Japanese Katakana)
3	%CharType_ZenkakuKatakana	Zenkaku Katakana (double-byte Japanese Katakana)
4	%CharType_Hiragana	Hiragana (Japanese)
5	%CharType_Kanji	Kanji (Japanese)
6	%CharType_DBAlphaNumeric	Double-byte Alphanumeric (Japanese)
7,8,9		Reserved for future use
10	%CharType_JapanesePunctuation	Japanese punctuation

Returns

ContainsOnlyCharType returns one of the following Number values. You can check for the constant instead of the numeric value, if you prefer:

Numeric Value	Constant Value	Description
1	%CharType_Matched	String contains only characters belonging to the sets listed in *char_code_list*.
0	%CharType_NotMatched	String contains one or more characters that do not belong to sets listed in *char_code_list*.
-1	%CharType_Unknown	UNKNOWN: unable to determine whether one or more characters in string is of set *char_code*. This occurs if the application is running on a non-Japanese version of Windows or on a UNIX application server when the CharacterSet parameter in the PSADMIN table is not set to SJIS.

Note. If any character is determined to be UNKNOWN, the return value is UNKNOWN.

Example

This example tests to see is the string is only Hiragana or Punctuation:

```
&ONLYHIRAGANA = ContainsOnlyCharType(&STRTOTEST, 4, 10);
If &ONLYHIRAGANA = 1 Then
   WinMessage("There are only Hiragana and Punctuation characters");
Else
   If &ONLYHIRAGANA = 0 Then
      WinMessage("Mixed characters");
   Else
      WinMessage("UNKNOWN");
   End-If
End-If
```

See Also

Chapter 1, PeopleCode Built-in Functions, CharType , page 64, Chapter 1, PeopleCode Built-in Functions, ContainsCharType , page 75, Chapter 1, PeopleCode Built-in Functions, ConvertChar , page 79, *PeopleTools Global Technology*, Character Sets and Language Input/Output.

ConvertChar

Syntax

ConvertChar(*source_str*, *source_str_category*, *output_str*, *target_char_code*)

Description

ConvertChar converts every character in *source_str* to type *target_char_code*, if possible, and places the converted string in *output_str*. ConvertChar supports the following conversion:

- Conversion among the Japanese Hankaku Katakana, Zenkaku Katakana, and Hiragana character sets.
- Conversion of Japanese Hankaku Katakana, Zenkaku Katakana, and Hiragana character sets to ASCII single-byte alphanumeric characters.

- Conversion of Japanese double-byte alphanumeric characters to their ASCII single-byte alphanumeric equivalents.
- Conversion of Japanese double-byte punctuation characters to their ASCII single-byte equivalents.

Other *source_str* and *target_char_code* combinations are either passed through without conversion, or not supported. Character types 0 and 1 (alphanumeric and extended Latin-1) are always passed through to *output_str* without conversion. See the Supported Conventions section later in this reference entry for details.

Japanese character sets can be recognized only if the application is running on a Japanese version of Windows, or, if on a UNIX application server, the CharacterSet parameter in the PSADMIN table is set to SJIS.

If ConvertChar is unable to determine whether the characters in *source_str* belong to the specified character set, the function returns a value of UNKNOWN (-1). This occurs if the application is running on a non-Japanese version of Windows or on a UNIX application server when the CharacterSet parameter in the PSADMIN table has not been set to SJIS. UNKNOWN characters are echoed as-is to the output string. If *source_str* can be partially converted, ConvertChar will partially convert string, echo the remaining characters to the output string as-is, and return a value of -2 (Completed with Issues).

Parameters

Parameter	Description
Source_str	String to be converted.
Source_str_category	Language category of input string. You can specify either a number or a constant.

Numeric Value	Constant Value	Description
0	%ConvertChar_AlphaNumeric	AlphaNumeric (7-bit ASCII codes: A-Z, a-z, 1-9, punctuation)
1	%ConvertChar_ExtendedLatin1	Extended Latin-1 Characters (ISO8859-1 accents, Spanish, French etc.)
2	%ConvertChar_Japanese	Japanese (any)

Parameter	Description
Output_str	A String variable to receive the converted string.
Target_char_code	Either a Number or a constant representing the conversion target character type. You can specify either a character code numeric value or a constant:

Numeric Value	Constant Value	Description
0	%CharType_AlphaNumeric	AlphaNumeric (7-bit ASCII codes: A-Z, a-z, 1-9, punctuation)
2	%CharType_HankakuKatakana	Hankaku Katakana (Single-byte Japanese Katakana)
3	%CharType_ZenkakuKatakana	Zenkaku Katakana (Double-byte Japanese Katakana)
4	%CharType_Hiragana	Hiragana (Japanese)
6	%CharType_DBAlphaNumeric	Double-byte AlphaNumeric (Japanese)

The following target values are not supported; if the source string is of the same type as any of these values, then the string is passed through without conversion.

Numeric Value	Constant Value	Description
1	%CharType_ExtendedLatin1	Extended Latin-1 characters (ISO8859-1 accents for Spanish, French, etc.)
5	%CharType_Kanji	Kanji (Japanese)
10	%CharType_JapanesePunctuation	Japanese punctuation

Supported Conversions

The following table shows which conversions are supported, which are passed through without conversion, and which are not supported:

Source	Target	Conversion
0 (Alphanumeric ASCII)	0-6 (All supported character types)	Pass through without conversion
1 (Extended Latin-1 characters)	0-6 (All supported character sets)	Pass through without conversion
2 (Hankaku Katakana)	0 (Alphanumeric)	Conversion supported
	1 (Extended Latin)	Not supported
	2 (Hankaku Katakana)	Pass through without conversion
	3 (Zenkaku Katakana)	Conversion supported
	4 (Hiragana)	Conversion supported
	5 (Kanji)	Not supported
	6 (Double-bye alphanumeric)	Not supported
3 (Zenkaku Katakana)	0 (Alphanumeric)	Conversion supported
	1 (Extended Latin)	Not supported
	2 (Hankaku Katakana)	Conversion supported
	3 (Zenkaku Katakana)	Pass through without conversion

Source	Target	Conversion
	4 (Hiragana)	Conversion supported
	5 (Kanji)	Not supported
	6 (Double-bye alphanumeric)	Not supported
4 (Hiragana)	0 (Alphanumeric)	Conversion supported
	1 (Extended Latin)	Not supported
	2 (Hankaku Katakana)	Conversion supported
	3 (Zenkaku Katakana)	Conversion supported
	4 (Hiragana)	Pass through without conversion
	5 (Kanji)	Not supported
	6 (Double-bye alphanumeric)	Not supported
5 (Kanji)	0-4, 6	Not supported
	5 (Kanji)	Pass through without conversion
6 (Double-byte alphanumeric)	0 (Alphanumeric)	Conversion supported
	1-5	Not supported
	6 (Double-bye alphanumeric)	Pass through without conversion
10 (Japanese punctuation)	0 (Alphanumeric)	Conversion supported
	1 (Extended Latin)	Not supported
	3-6, 10	Pass through without conversion

Returns

Returns either a Number or a constant with one of the following values, depending on what you’re checking for:

Numeric Value	Constant Value	Description
1	%ConvertChar_Success	String successfully converted.
0	%ConvertChar_NotConverted	String not converted.
-1	%ConvertChar_Unknown	UNKNOWN condition: this will occur if the application is running on a non-Japanese version of Windows or on a UNIX application server when the CharacterSet parameter of the PSADMIN table is not set to SJIS.
-2	%ConvertChar_Issues	Completed with issues. Conversion executed but there were one or more characters encountered that were either not recognized, or whose conversion is not supported.

Note. If any character cannot be translated, it is echoed as-is to *output_str*. *output_str* could therefore be a mixture of converted and non-converted characters.

Example

This example attempts to convert a string to Hiragana:

```
&RETVALUE = ConvertChar(&INSTR, 2, &OUTSTR, 4);
If &RETVALUE = 1 Then
   WinMessage("Conversion to Hiragana successful");
Else
   If &RETVALUE = 0 Then
      WinMessage("Conversion to Hiragana failed");
   Else
      If &RETVALUE = - 1 Then
         WinMessage("Input string is UNKNOWN character type.");
      Else
         WinMessage("Some characters could not be converted.");
      End-If
   End-If
End-If
```

See Also

Chapter 1, PeopleCode Built-in Functions, CharType , page 64, Chapter 1, PeopleCode Built-in Functions, ContainsCharType , page 75, Chapter 1, PeopleCode Built-in Functions, ContainsOnlyCharType , page 77, *PeopleTools Global Technology*, Character Sets and Language Input/Output.

ConvertCurrency

Syntax

ConvertCurrency(*amt*, *currency_cd*, *exchng_to_currency*, *exchng_rt_type*, *effdt*, *converted_amt* [, *error_process* [, *round*] [, *rt_index*]])

Description

Use the **ConvertCurrency** function to convert between currencies. The result of the conversion is placed in a variable passed in *converted_amt*.

See Also

PeopleTools Global Technology, Controlling Currency Display Format.

Parameters

Parameter	Description
Amt	The currency amount to be converted.
Currency_cd	The currency in which the *amt* is currently expressed.
Exchng_to_currency	The currency to which the *amt* should be converted.
Exchng_rt_type	The currency exchange rate to be used. This is the value of the RT_TYPE field in the RT_RATE table of RT_DFLT_VW.
Effdt	The effective date of the conversion to be used.
Converted_amt	The resulting converted amount. You must supply a variable for this parameter. If a conversion rate cannot be found, *converted_amt* is set equal to *amt*.
Error_process	An optional string that, if specified, contains one of the following values: • "F" - Produce a fatal error if a matching conversion rate is not found. • "W" - Produce a warning message box if a matching conversion rate is not found. • "I" - Or other−return without producing a message box If *error_process* is not specified, it defaults to Fatal ("F").
Round	Optional Boolean value indicating whether to round *converted_amt* to the smallest currency unit. If omitted, round defaults to False.
rt_index	An optional string to indicate which exchange rate index should be used to retrieve the exchange rate. If omitted, the Default Exchange Rate index (as specified on the Market Rate index definition) is used.

Note. If the currency exchange rate is changed in a PeopleSoft table, this change will *not* be reflected in an already open page until the user closes the page, then opens it again.

Returns

ConvertCurrency returns a Boolean value where True means a conversion rate was found and *converted_amt* calculated, and False means a conversion rate was not found and a value of one (1) was used.

Example

```
rem **-----------------------------------------------**;
rem *  Convert the cost & accum_depr fields if books *;
rem * use different currencies.  *;
rem **-----------------------------------------------**;
rem;
   If &FROM_CUR <> &PROFILE_CUR_CD Then
      &CON_COST_FROM = &COST_COST;
      &CON_ACC_DEPR_FROM = &COST_ACCUM;
      ConvertCurrency(&CON_COST_FROM, &FROM_CUR, &PROFILE_CUR_CD, RT_TYPE,⇒
TRANS_DT, &CON_COST_TO, "F");
```

```
      UpdateValue(COST_NON_CAP.COST, &COST_ROW_CUR, &CON_COST_TO);
   Else
      UpdateValue(COST_NON_CAP.COST, &COST_ROW_CUR, &COST_COST);
   End-If;
   UpdateValue(COST_NON_CAP.FROM_CUR, &COST_ROW_CUR, &PROFILE_CUR_CD);
UpdateValue(COST_NON_CAP.OPRID, &COST_ROW_CUR, %UserIdId);
```

ConvertDatetimeToBase

Syntax

ConvertDatetimeToBase(*textdatetime*, {*timezone* | **"Local"** | **"Base"**});

Description

ConvertDatetimeToBase converts the text value *textdatetime* to a datetime value, then further converts it to the base time. This function automatically calculates whether daylight savings time is in effect for the given *textdatetime*.

The system's base time zone is specified on the PSOPTIONS table.

Parameters

Parameter	Description
textdatetime	Specify a datetime value represented as text (for example, "01/01/99 3:00 PM")
timezone | **Local** | **Base**	Specify a value for converting *textdatetime*. Values are: • *timezone* - a time zone abbreviation or a field reference to be used for converting *textdatetime* • **Local** - use the local time zone for converting *textdatetime*. • **Base** - use the base time zone for converting *textdatetime*.

Returns

Returns a datetime value in the base time zone.

Example

In the following example, &DATETIMEVAL would have a datetime value of "01-01-1999-07:00:00.000000", assuming the Base time (as defined in PSOPTIONS) was PST.

```
&DATETIMEVAL= ConvertDateTimeToBase("01/01/99 10:00:00AM", "EST");
```

See Also

Chapter 1, PeopleCode Built-in Functions, ConvertTimeToBase , page 87, Chapter 1, PeopleCode Built-in Functions, FormatDateTime , page 192, Chapter 1, PeopleCode Built-in Functions, IsDaylightSavings , page 302, Chapter 1, PeopleCode Built-in Functions, DateTimeToTimeZone , page 124, Chapter 1, PeopleCode Built-in Functions, TimeToTimeZone , page 460, Chapter 1, PeopleCode Built-in Functions, TimeZoneOffset , page 461, *PeopleSoft Server Tools Administration*, PeopleTools Utilities.

ConvertRate

Syntax

ConvertRate(*Rate, In_Frequency, Out_Frequency*)

Description

The **ConvertRate** function converts a rate between various compounding frequencies.

Parameters

Parameter	Description
Rate	The rate to be converted. This parameter takes a number value.
In_Frequency	The frequency of the rate to be converted from. This parameter takes an array of number, with two elements. The first element is periodicity, (for example, if you chose daily compounding, 1 would represent daily while 7 would represent weekly.) The second element is the unit of measure of frequency. The values for the second element are:

Value	Description
0	continuous compounding
1	daily compounding
2	monthly compounding
3	yearly compounding

Parameter	Description
Out_Frequency	The frequency of the rate to be converted to. This parameter takes an array of number, with two elements. The first element is periodicity, (for example, if you chose daily compounding, 1 would represent daily while 7 would represent weekly.) The second element is the unit of measure of frequency. The values for the second element are:

Value	Description
0	continuous compounding
1	daily compounding
2	monthly compounding
3	yearly compounding

Returns

A number representing the converted rate.

Example

The following example converts the specified values from days to years.

```
Local array of number &In, &Out;
Local number &rate, &NewRate;

&rate = 0.01891;
&In = CreateArray(0, 0);
&In[1] = 1; /* daily */
&In[2] = 1; /* compound_days */
&Out = CreateArray(0, 0);
&Out[1] = 1; /* one year */
&Out[2] = 3; /* compound_years */

&NewRate = ConvertRate(&rate, &In, &Out);
```

See Also

Chapter 1, PeopleCode Built-in Functions, RoundCurrency , page 382.

ConvertTimeToBase

Syntax

ConvertTimeToBase(*texttime*,{*timezone* | **"Local"** | **"Base"**});

Description

ConvertTimeToBase converts the text value *texttime* to a Time value and converts it to the base time. This function automatically calculates whether daylight savings time is in effect for the given *texttime*.

This function is useful for users to convert constant times in specific time zones into database time. For example, there is a deadline for completing Federal Funds transfers by 3:00 PM Eastern Time. ConvertTimeToBase does this conversion, taking into account daylight savings time. The date used to calculate whether daylight savings time is in effect is the current date.

The system's base time zone is specified on the PSOPTIONS table.

Parameters

Parameter	Description
texttime	Specify a time value represented as text (e.g., "3:00 PM")
timezone | **Local** | **Base**	Specify a value for converting *textdatetime*. Values are: • *timezone* - a time zone abbreviation or a field reference to be used for converting *textdatetime* • **Local** - use the local time zone for converting *textdatetime*. • **Base** - use the base time zone for converting *textdatetime*.

Returns

Returns a time value in the base time zone.

Example

In the following example, &TIMEVAL would have a time value of "07:00:00.000000", assuming the Base time (as defined in PSOPTIONS) was PST.

```
&TEXTTIME = ConvertTimeToBase("01/01/99 10:00:00AM", "EST");
```

See Also

Chapter 1, PeopleCode Built-in Functions, ConvertDatetimeToBase , page 85, Chapter 1, PeopleCode Built-in Functions, FormatDateTime , page 192, Chapter 1, PeopleCode Built-in Functions, IsDaylightSavings , page 302, Chapter 1, PeopleCode Built-in Functions, DateTimeToTimeZone , page 124, Chapter 1, PeopleCode Built-in Functions, TimeToTimeZone , page 460, Chapter 1, PeopleCode Built-in Functions, TimeZoneOffset , page 461, *PeopleSoft Server Tools Administration*, PeopleTools Utilities.

CopyAttachments

Syntax

CopyAttachments(*URLSource*, *URLDestination* [, *Record_Names*])

where *URLSource* and *URLDestination* can have *one* of the following forms:
`URL.URLname`

or a string URL, such as
`ftp://user:password@ftp.ps.com/`

Description

Use the **CopyAttachments** function to copy files from one system to another.

Note. If the specified subdirectories do not exist this function tries to create them.

Parameters

Parameter	Description
Source	Specify the source location of the files to be moved. This parameter can either be a URL name in the form **URL** .*URLName*, or a string.
Destination	Specify the destination of the files being moved. This parameter can either be a URL name in the form **URL** .*URLName*, or a string.
Record_Names	Specify an array of record names that store valid references to files located on the system specified by *Source*. If you don't specify this parameter, all the records that contain the field ATTACHSYSFILENAME are used (that is, this function tries to get the attachments based on all those references from *Source*).

Returns

You can check for either a constant or a number value. The values are:

Numeric Value	Constant Value	Description
0	%Attachment_Success	File was transferred successfully.
1	%Attachment_Failed	File was not successfully transferred.

Numeric Value	Constant Value	Description
3	%Attachment_FileTransferFailed	File transfer didn't succeed.
7	%Attachment_DestSystNotFound	Cannot locate destination system.
8	%Attachment_DestSystFailedLogin	Unable to login to destination system.
9	%Attachment_FileNotFound	Cannot locate file.
10	%Attachment_DeleteFailed	Cannot delete file.

Example

```
&aRecs = CreateArray("HRATTS", "MFGATTS", "CRMATTS");

&ret = CopyAttachments("ftp://user:pass@system/HR/", "record://HRARCHIVE", ⇒
&aRecs);

If (&ret = %Attachment_Success) Then
   MessageBox(0, "Copy Archive Status", 0, 0, "CopyAttachmentArchive succeeded");
Else
   MessageBox(0, "Copy Archive Status", 0, 0, "CopyAttachmentArchivefailed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, AddAttachment , page 31, Chapter 1, PeopleCode Built-in Functions, DeleteAttachment , page 135, Chapter 1, PeopleCode Built-in Functions, GetAttachment , page 234, Chapter 1, PeopleCode Built-in Functions, PutAttachment , page 353, Chapter 1, PeopleCode Built-in Functions, ViewAttachment , page 494, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Understanding File Attachment Architecture, *PeopleSoft Server Tools Administration*, PeopleTools Utilities, Copy File Attachments.

CopyFields

Syntax

CopyFields(*from, to*)

where *from* and *to* are constructions that reference rows of data on specific source and target records in the component buffer; each have the following syntax:

level, scrollpath, target_row

and where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD.***target_recname*

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

CopyFields copies like-named fields from a row on the specific source record to a row on the specific target record.

Note. This function remains for backward compatibility only. Use the **CopyFieldsTo** or **CopyChangedFieldsTo** record class methods instead.

See Also

Chapter 25, Record Class , CopyFieldsTo, page 1445, Chapter 25, Record Class , CopyChangedFieldsTo, page 1444, Chapter 26, Row Class , CopyTo, page 1467, Chapter 27, Rowset Class , CopyTo, page 1482, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
from	A placeholder for a construction (*level*, *scrollpath*, *target_row*) that references the first row in the comparison.
to	A placeholder for a construction (*level*, *scrollpath*, *target_row*) that references the second row in the comparison.
level	Specifies the scroll level for the target level scroll.
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	Specifies the row number of each target row on its scroll level.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Example

The following example copies fields from PO_RECEIVED_INV (level 1 scroll) from row &ROW to PO_RECV_INV_VW (level 1 scroll), row &LOC_ROW:

```
CopyFields(1, RECORD.PO_RECEIVED_INV, &ROW, 1, RECORD.PO_RECV_INV_VW, &LOC_ROW);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CompareLikeFields , page 73, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

CopyRow

Syntax

CopyRow(*destination_row*, *source_row*)

Description

CopyRow copies the data from one row to another row.

Note. This function remains for backward compatibility only. Use the **CopyTo** Row class method instead.

destination_row is the row number to which you want the *source _row* data values copied. The two rows, and the PeopleCode function call, must all be located on the same scroll level.

See Also

Chapter 26, Row Class , CopyTo, page 1467, Chapter 27, Rowset Class , CopyTo, page 1482, Chapter 25, Record Class , CopyFieldsTo, page 1445, Chapter 25, Record Class , CopyChangedFieldsTo, page 1444, *PeopleCode Developer's Guide*, Data Buffer Access.

Parameters

Parameter	Description
destination_row	Row number of row to which to copy data.
source_row	Row number of row from which to read data.

Example

This example uses **CopyRow** to give an inserted row the same values as the previous row:

```
/*  Get the row number of the inserted row and the previous row */
&NEW_ROW_NUM = CurrentRowNumber();
&LAST_ROW_NUM = &NEW_ROW_NUM - 1;
/*  Copy the data from the previous row into the inserted row */
CopyRow(&NEW_ROW_NUM, &LAST_ROW_NUM);
```

Cos

Syntax

Cos(*angle*)

Description

Cos calculates the cosine of the given angle (adjacent / hypotenuse).

Parameters

Parameter	Description
angle	A value in radians.

Returns

A real number between -1.00 and 1.00.

Example

The following example returns the cosine of an angle measuring 1.2 radians:

```
&MY_RESULT = Cos(1.2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Acos , page 29, Chapter 1, PeopleCode Built-in Functions, Asin , page 45, Chapter 1, PeopleCode Built-in Functions, Atan , page 46, Chapter 1, PeopleCode Built-in Functions, Cot , page 92, Chapter 1, PeopleCode Built-in Functions, Degrees , page 134, Chapter 1, PeopleCode Built-in Functions, Radians , page 356, Chapter 1, PeopleCode Built-in Functions, Sin , page 439, Chapter 1, PeopleCode Built-in Functions, Tan , page 456.

Cot

Syntax

Cot(*angle*)

Description

Cot calculates the cotangent of the given angle (adjacent / opposite, that is, the reciprocal of the tangent).

Parameters

Parameter	Description
angle	A value in radians, excluding 0. If the input value is 0, you receive an error message at runtime (“Decimal arithmetic error occurred. (2,110)”). Adjust your code to provide a valid input value.

Note. In theory, all values of *angle* such that *angle* mod pi = 0 are not valid for this function, because inputs approaching such values produce results that tend toward infinity. In practice, however, no computer system can represent such values exactly. Thus, for example, the statement `Cot(Radians(180))` produces a number close to the largest value PeopleCode can represent, rather than an error.

Returns

A real number.

Example

The following example returns the cotangent of an angle measuring 1.2 radians:

```
&MY_RESULT = Cot(1.2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Acos , page 29, Chapter 1, PeopleCode Built-in Functions, Asin , page 45, Chapter 1, PeopleCode Built-in Functions, Atan , page 46, Chapter 1, PeopleCode Built-in Functions, Cos , page 91, Chapter 1, PeopleCode Built-in Functions, Degrees , page 134, Chapter 1, PeopleCode Built-in Functions, Radians , page 356, Chapter 1, PeopleCode Built-in Functions, Sin , page 439, Chapter 1, PeopleCode Built-in Functions, Tan , page 456.

CreateArray

Syntax

CreateArray(*paramlist*)

Where *paramlist* is an arbitrary-length list of values in the form:

param1 [, *param2*] ...

Description

CreateArray constructs an array and returns a reference to it.

The type of the first parameter determines the type of array that is built. That is, if the first parameter is of type NUMBER, an array of number is built. If there is no first parameter an empty array of ANY is built.

The CreateArray function uses flattening and promotion as required to convert subsequent values into suitable elements of the array.

Parameters

Parameter	Description
paramlist	Specify a list of values to be used as the elements of the array.

Returns

Returns a reference to the array.

Example

```
Local Array of Array of Number &AAN;
Local Array of Number &AN;

&AAN = CreateArray(CreateArray(1, 2), CreateArray(3, 4), 5);
&AN = CreateArray(6, &AAN[1]);
```

&AAN is a two dimensional array with three elements:

- A one-dimensional array with 1 and 2 as elements.
- A one-dimensional array with 3 and 4.
- A one-dimensional array with only the element 5.

The last parameter to Array was promoted to a one-dimensional array. &AN is a one-dimensional array with 3 elements: 6, 1, and 2. The last parameter to Array in the last line was flattened into its two elements.

See Also

Chapter 1, PeopleCode Built-in Functions, CreateArrayRept , page 94, Chapter 1, PeopleCode Built-in Functions, Split , page 441, Chapter 7, Array Class , page 633, Chapter 7, Array Class , Using Flattening and Promotion , page 640.

CreateArrayAny

Syntax

CreateArrayAny([*paramlist*])

Where *paramlist* is an arbitrary-length list of values in the form:
param1 [, *param2*] ...

Description

CreateArrayAny constructs an array whose elements are of data type ANY and returns a reference to it.

The CreateArrayAny function uses flattening and promotion as required to convert subsequent values into suitable elements of the array.

If you don't specify any parameters with CreateArrayAny, it's the same as using the **CreateArray** function without any parameters.

If you don't know how many values are needed in a SQL statement until runtime, you can use an array of any to supply the values.

Parameters

Parameter	Description
paramlist	Specify a list of values to be used as the elements of the array.

Returns

Returns a reference to an array of ANY.

Example

```
local Array of Any &ArrayAny = CreateArrayAny(1, 2, "hi", "there");

local Array of Array of Any &AAAny = CreateArray(CreateArrayAny(1, 2), ⇒
CreateArrayAny("hi"), "there");
```

&ArrayAny is a two dimensional array with four elements: 1, 2, "hi" and "there". All the elements have the data type Any.

&AAAny is a two-dimensional array with three elements: a one-dimensional array with 1 and 2 as elements, a one-dimensional array with "hi" as its element, and a one-dimensional array with only the element "there". The last parameter to the array was promoted to a one-dimensional array.

See Also

Chapter 1, PeopleCode Built-in Functions, CreateArrayRept , page 94, Chapter 1, PeopleCode Built-in Functions, CreateArray , page 92, Chapter 1, PeopleCode Built-in Functions, Split , page 441, Chapter 7, Array Class , page 633, Chapter 7, Array Class , Using Flattening and Promotion , page 640, Chapter 31, SQL Class , page 1613.

CreateArrayRept

Syntax

CreateArrayRept(*val, count*)

Description

CreateArrayRept creates an array that contains *count* copies of *val*. If *val* is itself an array, the created array has one higher dimension, and each element (sub-array) is the array reference *val*.

The type of the first parameter (*val*) determines the type of array that is built. That is, if the first parameter is of type NUMBER, an array of number is built. If *count* is zero, CreateArrayRept creates an empty array, using the *val* parameter for the type.

If you are making an array that is multi-dimensional, *val* will be the subarray used as the elements.

To make the subarrays distinct, use the Clone method. For example:

```
&A = CreateArrayRept(&AN, 4).Clone();
```

Parameters

Parameter	Description
val	A value of any type.
count	The number of copies of *val* contained in the array.

Returns

Returns a reference to the array.

Example

The following code sets &A to a new empty array of string:

```
&A = CreateArrayRept("", 0);
```

The following code sets &A to a new array of ten zeroes:

```
&A = CreateArrayRept(0, 10);
```

The following code sets &AAS to a new array of array of strings, with two subarrays:

```
&AAS = CreateArrayRept(CreateArray("one", "two"), 2);
```

Note that in this case, both elements (rows) of &AAS contain the *same* subarray, and changing the value of an element in one of them changes it in *both* of them. To get distinct subarrays, use the Clone method:

```
&AAS = CreateArrayRept(CreateArray("one", "two"), 2).Clone();
```

The following example shows how to create a two-dimension array using CreateArrayRept and Push. In addition, it shows how to randomly assigns values to the cells in a two-dimension array.

```
Local array of array of string &ValueArray;

&Dim1 = 10;
&Dim2 = 5;
&ValueArray = CreateArrayRept(CreateArrayRept("", 0), 0);
For &I = 1 To &Dim1
   &ValueArray.Push(CreateArrayRept("", &Dim2));
End-For;
&ValueArray[1][1] = "V11";
&ValueArray[2][1] = "V21";

WinMessage("&ValueArray[1][1] = " | &ValueArray[1][1] | " " | "&ValueArray[2][1] =⇒
```

```
" | &ValueArray[2][1], 0);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateArray , page 92, Chapter 1, PeopleCode Built-in Functions, Split , page 441, Chapter 7, Array Class , page 633, Chapter 7, Array Class , Clone, page 641.

CreateDirectory

Syntax

CreateDirectory(*path*, [*, pathtype*])

Description

The **CreateDirectory** function creates the directory specified by *path* and any non-existent directories specified in *path*.

On UNIX systems, the directory has the mode 755, that is, read-write-execute permission for the owner, while group and other have only read and execute permission.

```
drwxr-xr-x
```

Working with Relative Paths

If you specify a relative path, that path is appended to the path specified by an environmental variable, if the variable exists, in a particular order. This means if the first environmental variable in order is found, that is the one that's used. If it isn't found, the next one is searched for, and if found, is used, and so on.

The environmental variables that get used are different, depending on whether the machine where the code is running is an application server or not. If the code is running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]

If the code is *not* running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]
3. TEMP[\relative_path]

The PS_FILEDIR environmental variable is *not* initialized and set to a path automatically. To use this environmental variable, you must set it yourself.

Note. Your system security should verify if a user has the correct permissions before allowing access to a drive (for example, if a user changes their **TEMP** environmental variable to a network drive they don't normally have access to, your system security should detect this.)

Parameters

Parameter	Description
path	Specify the path to be created.
pathtype	Specify whether the path is an absolute or relative path. The values for this parameter are: • %FilePath_Relative (default) • %FilePath_Absolute If you don't specify *pathtype* the default is %FilePath_Relative. If *pathtype* is an absolute path, whatever path you specify is used verbatim. You *must* specify a drive letter and the complete path. You can't use any wildcards when specifying a path. The PeopleCode runtime automatically converts platform-specific separator characters to the appropriate form for where your PeopleCode program is executing. On a WIN32 system, UNIX "/" separators are converted to "\", and on a UNIX system, WIN32 "\" separators are converted to "/".

Note. The syntax of the path only depends on the platform where your PeopleCode is executing.

Returns

None.

Example

```
CreateDirectory("D:\Resumes\New_Hires", %FilePath_Absolute);
```

See Also

Chapter 1, PeopleCode Built-in Functions, FileExists , page 184, Chapter 1, PeopleCode Built-in Functions, FindFiles , page 189, Chapter 1, PeopleCode Built-in Functions, GetFile , page 243, Chapter 1, PeopleCode Built-in Functions, GetAttachment , page 234, Chapter 1, PeopleCode Built-in Functions, PutAttachment , page 353, Chapter 1, PeopleCode Built-in Functions, RemoveDirectory , page 367, Chapter 13, File Class , page 873.

CreateException

Syntax

CreateException(*message_set*, *message_num*, *default_txt [*, *subslist*])

where *subslist* is an arbitrary-length list of substitutions of undetermined (Any) data type to be substituted in the resulting text string, in the form:
substitution1 [, *substitution2*]. . .

Description

The **CreateException** function creates an exception object with the given properties. You can use this in your exception handling. Use this function either in conjunction with the Throw statement, or on its own to get more information of a message.

Parameters

Parameter	Description
message_set	Specify the message set number of the message you want associated with this exception.
message_num	Specify the message number of the message you want associated with this exception.
default_txt	Specify the text you want associated with the exception if the message specified by *message_set* and *message_num* isn't found.
subslist	A comma-separated list of substitutions; the number of substitutions in the list is arbitrary. The substitutions are referenced in the message text using the % character followed by an integer corresponding to the position of the substitution in the *subslist*.

Returns

A reference to an exception object if successful, NULL otherwise.

Example

```
Function t2
   throw CreateException(2, 160, "'%1' doesn't support property or method '%2'",⇒
 "SomeClass", "SomeMethod");
End-Function;
```

See Also

Chapter 11, Exception Class , page 835.

CreateJavaArray

Syntax

CreateJavaArray(*ElementClassName*[], *NumberOfElements*)

Description

The **CreateJavaArray** function enables you to create a Java array without knowing the number or value of the elements.

When you create an array in Java, you already know the number of elements in the array. If you don't know the number of elements in the array, but you want to use a Java array, use the CreateJavaArray function in PeopleCode. This creates a Java object that is a Java array, and you can pass in the number of elements that are to be in the array.

The first index in a Java array is 0. PeopleCode arrays start at 1.

Do the following to specify this type of array in Java:

```
new ElementClassName[NumberOfElements];
```

Parameters

Parameter	Description
ElementClassName[]	Specify the array class name. This parameter takes a string value.
NumberOfElements	Specify the number of elements in the array. This parameter takes a number value.

Returns

A Java object

See Also

Chapter 1, PeopleCode Built-in Functions, CreateJavaObject , page 99, Chapter 1, PeopleCode Built-in Functions, GetJavaClass , page 252, Chapter 16, Java Class, page 959.

CreateJavaObject

Syntax

```
CreateJavaObject(ClassName [ConstructorParams])
```

Where *ConstructorParams* has the form

```
Arguement1 [, arguement2] . . .
```

Description

The **CreateJavaObject** function creates a Java object that can be manipulated in PeopleCode.

Note. If you create a class that you want to call using CreateJavaObject, it must be located in a directory specified in the PS_CLASSPATH environment variable or in the <PS_HOME>/appserv/classes directory.

Use the CreateJavaObject function to create a Java array when you know how many values it should contain. If *ClassName* is the name of an array class (ending with [ ]), *ConstructorParams* are used to initialize the array.

In Java, do the following to initialize an array:

```
intArray = new int[]{1, 2, 3, 5, 8, 13};
```

Do the following to initialize such a Java array from PeopleCode:

```
&IntArray = CreateJavaObject("int[]", 1, 2, 3, 5, 8, 13);
```

To initialize a Java array without knowing the number of parameters until runtime, use the CreateJavaArray function.

Parameters

Parameter	Description
ClassName	Specify the name of an already existing class.
ConstructorParams	Specify any construction parameters required for the class. Constructors are matched by construction parameter type and placement.

Returns

A Java object.

Example

The following is an example of using dot notation and CreateJavaObject.

```
&CHARACTER.Value = CreateJavaObject(&java_path).GetField(&java_newchar).Value;

&NUMBER.Value = CreateJavaObject(&java_path).GetField(&java_newnum).Value;

&DATE.Value = CreateJavaObject(&java_path).GetField(&java_newdate).Value;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateJavaArray , page 98, Chapter 1, PeopleCode Built-in Functions, GetJavaClass , page 252, Chapter 16, Java Class, page 959.

CreateMessage

Syntax

CreateMessage(**MESSAGE**.*messagename*)

Description

CreateMessage instantiates a message object that refers to a message definition created in PeopleSoft Application Designer. The CreateMessage function sets the following properties for the resulting message object, based on the values set for the message definition:

- Name
- ChannelName
- Active

Other properties are set when the message is published or subscribed to (PubID, SubName, and so on,) or are dynamically generated at other times (Size, EditError, and so on.)

CreateMessage also sets the LANGUAGE_CD field in the level 0 PSCAMA record for a message based on the USERID default language group. If the message is being published from a component, the language code is set to the USERID language code for the component. If CreateMessage is called from an Application Engine program, the language code of the user who started the batch process is used.

Parameters

Parameter	Description
MESSAGE. *messagename*	Specify the name of the message definition you want to create a message object for.

Returns

Returns a reference to a message object.

Example

The following example creates a message &MSG based on the message definition PURCHASE_ORDER.

```
Local message &MSG;

&MSG = CreateMessage(MESSAGE.PURCHASE_ORDER);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetMessage , page 255, Chapter 1, PeopleCode Built-in Functions, GetPubContractInstance , page 263, Chapter 1, PeopleCode Built-in Functions, GetSubContractInstance , page 274, Chapter 17, Message Class , page 973.

CreateObject

Syntax

CreateObject(*str_class_name, create_par, . . .*)

Where *str_class_name* either:

—identifies a class by class name

—identifies a class of OLE Automation object in the form:
app_name.object_name

Description

Use **CreateObject** to return an instance of a class. You can use this function to access an Application Class, a PeopleCode built-in object (like a chart), or an OLE Automation object.

If the class you are creating requires values to be passed, use the *create_par* parameters to supply them, or use the **CreateObjectArray** function.

Considerations Using Application Classes

You can use the CreateObject function to access an Application Class. You would want to do this when you were programming at a high-level, when you might not know the name of the class you wanted to access until runtime. You must specify a fully-qualified class name. In addition, the class name is case-sensitive.

The returned object has the type of class you specified.

Considerations Using PeopleCode Built-in Objects

For example, to generate a Peoplesoft chart object without using a chart control (that is, without using the **GetChart** function.) you could use:

```
&MyChart = CreateObject("Chart");
```

The returned object has the type of class you specified.

Considerations Using OLE Automation Objects

CreateObject returns an instance of an OLE Automation object as a variable of type Object.

The *str_class_name* argument uses the syntax *app_name.object_type*, which consists of: *app_name* (the name of the application providing the object) and *object_type* (the class or type of the object to create), separated by a period (dot).

Any application that supports OLE Automation exposes at least one type of object. For example, a spreadsheet application may provide an application object, a worksheet object, and a toolbar object.

To create an OLE Automation object, you assign the object returned by CreateObject to a variable of type Object:

```
local object &WORKSHEET;

&WORKSHEET = CreateObject("Excel.Sheet");
```

After an object is created, you can reference it using the object variable. In the previous example, you access properties and methods of the new object using the **ObjectGetProperty**, **ObjectSetProperty**, and **ObjectDoMethod** functions.

Note. If an object has registered itself as a single-instance object, only one instance of the object can be created, even if CreateObject is executed more than once. Note also that an object assigned to a global variable is not valid across processes: that is, the scope and lifetime of the global is the same as the scope and lifetime of the instance of PeopleTools in which the object was created.

Parameters

Parameter	Description
str_class_name	Specify the name of the class that you want to instantiate an object from.
create_par	Specify the parameters required by the class for instantiating the object.

Example

This example instantiates an Excel worksheet object, makes it visible, names it, saves it, and displays its name. Note the use of ObjectGetProperty in the example to return the Excel.Sheet.Application object.

```
&WORKAPP = CreateObject("COM", "Excel.Application");
&WORKBOOKS = ObjectGetProperty(&WORKAPP, "Workbooks");
ObjectDoMethod(&WORKBOOKS, "Add", "C:\TEMP\INVOICE.XLT"); /* This associates the⇒
 INVOICE template w/the workbook */
ObjectDoMethod(&WORKAPP, "Save", "C:\TEMP\TEST1.XLS");
ObjectSetProperty(&WORKAPP, "Visible", True);
```

This following example illustrates the creation of an application class object. This code assumes that MyBaseClass is the superclass of both MySubclass1 and MySubclass2 classes.

```
local MyBaseClass &mbobj;
local String &ClassName = "MySubclass1";
if &test then
&ClassName = "MySubclass2";
end-if;
&mbobj = CreateObject(&ClassName);
```

The following example creates a chart in an iScript, using the CreateObject function to generate a reference to a chart object.

```
Function IScript_GetChartURL()

   Local Chart &oChart;
```

```
   Local Rowset &oRowset;
   Local string &sMap;
   Local string &sURL;

   &oChart = CreateObject("Chart");

   &oRowset = CreateRowset(Record.QE_CHART_RECORD);
   &oRowset.Fill("where QE_CHART_REGION= :1", "MIDWEST");
   &oChart.SetData(&oRowset);

   &oChart.Width = 400;
   &oChart.Height = 300;

   &oChart.SetDataYAxis(QE_CHART_RECORD.QE_CHART_SALES);
   &oChart.SetDataXAxis(QE_CHART_RECORD.QE_CHART_PRODUCT);
   &oChart.SetDataSeries(QE_CHART_RECORD.QE_CHART_REGION);

   &oChart.HasLegend = True;
   &oChart.LegendPosition = %ChartLegend_Right;

   &sURL = %Response.GetChartURL(&oChart);
   &sMap = &oChart.ImageMap;

   %Response.Write("<HTML><IMG SRC=");
   %Response.Write(&sURL);
   %Response.Write("  USEMAP=#THEMAP></IMG><MAP NAME=THEMAP>");
   %Response.Write(&sMap);
   %Response.Write("</MAP></HTML>");

End-Function;
```

See Also

Chapter 1, PeopleCode Built-in Functions, ObjectDoMethod , page 340, Chapter 1, PeopleCode Built-in Functions, ObjectGetProperty , page 342, Chapter 1, PeopleCode Built-in Functions, ObjectSetProperty , page 343, Chapter 1, PeopleCode Built-in Functions, CreateObjectArray , page 103, Chapter 1, PeopleCode Built-in Functions, ObjectDoMethodArray , page 341, Chapter 15, Internet Script Classes (iScript) , page 931, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using OLE Functions.

CreateObjectArray

Syntax

CreateObjectArray(*Class_Name*, *Array_of_Args*)

Description

Use the **CreateObjectArray** function to return an instance of a class.

Use this function when you must pass in parameters to create the object and you don't know when you write the code how many parameters are required. If you can create the object without passing in additional values, or if you know how many parameters are required, use the **CreateObject** function instead.

The array of parameters is an array of Any. It *must* be a one-dimensional array, that is, you cannot pass in an array of array of Any. You cannot pass in field references, that is, you cannot pass in references of the form:

```
RECORD.FIELDNAME
```

If you do not want to supply any parameters, you can use an empty array, or a reference to a Null array.

Parameters

Parameter	Description
Class_Name	Specify the name of the class you want to create an instance of, as a string.
Array_Of_Args	Specify an Array of Any containing all parameters for creating an instance of the class.

Returns

A reference to newly created object.

Example

The following is an example of the creation of an Application Class object where the number of parameters used to create the object varies, depending on data in the database.

```
local String &ClassName, &RecName;
local Record &Rec;

/* Read class name and parameter record name from the database. */
SQLExec("SELECT CLASSNAME, RECNAME FROM %TABLE(RECORD.CLASSDATA)", &ClassName,⇒
 &RecName);

local Record &Rec = CreateRecord(@ ("RECORD." | &RecName));

/* Read the parameters from the database. */
local Array of Any &Params = CreateArrayAny();

SQLExec("%SelectAll(:1)", &Rec, &Params);

/* Create the object. */
local MyPackage:BaseClass &Obj = CreateObjectArray(&ClassName, &Params);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateObject , page 101, Chapter 1, PeopleCode Built-in Functions, ObjectDoMethod , page 340, Chapter 1, PeopleCode Built-in Functions, ObjectGetProperty , page 342, Chapter 1, PeopleCode Built-in Functions, ObjectSetProperty , page 343, Chapter 1, PeopleCode Built-in Functions, ObjectDoMethodArray , page 341, Chapter 7, Array Class , page 633, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using OLE Functions.

CreateProcessRequest

Syntax

CreateProcessRequest([*ProcessType*, *ProcessName*])

Description

The **CreateProcessRequest** function creates a ProcessRequest object. After you've created this object, you can assign values to its properties then use the Schedule method to submit the process request for scheduling.

If you specify PSJob for the process type, the ProcessRequest object contains all the items of the job.

Parameters

Parameter	Description
ProcessType	Specify the process type as a string. Values depend on the process types defined for your system.
ProcessName	Specify the name of the process as a string.

Returns

A reference to a ProcessRequest object.

Example

```
Local ProcessRequest &MYRQST;

&MYRQST = CreateProcessRequest("PSJOB", &MyJobName);
```

See Also

Chapter 23, ProcessRequest Class , page 1269, Chapter 23, ProcessRequest Class , Schedule, page 1286, .

CreateRecord

Syntax

CreateRecord(**RECORD**.*recname*)

Description

CreateRecord creates a *standalone* record definition and its component set of field objects. The specified record must have been defined previously, that is, it must have a record definition. However, if you are calling this function from PeopleCode associated with a page, the record does not have to be included on the current page.

The record and field objects created by this function are accessible only within PeopleCode. They can be used with any of the record and field object methods and properties. The record and field objects are automatically deleted when there are no remaining references to them stored in any variables.

The fields created by this function are initialized to null values. Default processing is not performed. No data associated with the record definition's SQL table is brought in: only the record definition.

You can select into a record object created this way using the **SelectByKey** method.
You can also select into it using **SQLExec**.

Parameters

Parameter	Description
RECORD. *recname*	Specify a record definition that already exists.

Returns

This function returns a record object that references a new record buffer and set of fields.

Example

```
Local Record &REC2;

&REC2 = CreateRecord(RECORD.OPC_METH);
```

In the following example, a free-standing record is created (&PSBATREPREQRES). Values are assigned to the fields associated with the record. Then a second record is created (&PUBHDR), and the values from the first record are used to populate the second record.

```
&PSBATREPREQRES = CreateRecord(RECORD.PSBATREPREQRES);
   &PSBATREPREQRES.BATREPID.Value = &BATREPID;
   &PSBATREPREQRES.PUBID.Value = &MSG.Pubid;
   &PSBATREPREQRES.CHNLNAME.Value = &MSG.ChannelName;
   &PSBATREPREQRES.PUBNODE.Value = &MSG.PubNodeName;
   &PSBATREPREQRES.MSGNAME.Value = &MSG.Name;

   &PUBHDR = CreateRecord(RECORD.PSAPMSGPUBHDR);
   &PSBATREPREQRES.CopyFieldsTo(&PUBHDR);
```

To create a PeopleCode record object for a record whose name is unknown when the PeopleCode is written, do the following.

Suppose a record name is in the PeopleCode variable &RECNAME. Use the @ operator to convert the string to a component name. The following code creates a corresponding record object:

```
&RECNAME = "RECORD." | Upper(&RECNAME);
&REC = CreateRecord(@ &RECNAME);
```

The following example uses SQLExec to select into a record object, based on the effective date.

```
Local Record &DST;

&DST = CreateRecord(RECORD.DST_CODE_TBL);
&DST.SETID.Value = GetSetId(FIELD.BUSINESS_UNIT, DRAFT_BU, ⇒
RECORD.DST_CODE_TYPE, "");
&DST.DST_ID.Value = DST_ID_AR;
SQLExec("%SelectByKeyEffDt(:1,:2)", &DST, %Date, &DST);
/* do further processing using record methods and properties */
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetRecord , page 265, Chapter 1, PeopleCode Built-in Functions, GetField , page 242, Chapter 25, Record Class , page 1441, *PeopleCode Developer's Guide*, Data Buffer Access.

CreateRowset

Syntax

```
CreateRowset({RECORD.recname | &Rowset} [, {FIELD.fieldname,
RECORD.recname | &Rowset}] .  .  .)
```

Description

The **CreateRowset** function creates an *unpopulated*, standalone rowset.

A standalone rowset is a rowset that has the specified structure, but is not tied to any data (that is, to the component buffer or to a message.) In addition, a standalone rowset isn't tied to the Component Processor. When you fill it with data, no PeopleCode runs (like RowInsert, FieldDefault, and so on.)

The first parameter determines the structure of the rowset to be created.

If you specify a record as the first parameter, it's used as the primary level 0 record. If you don't specify any other parameters, you create a rowset containing one row, with one unpopulated record. To populate this type of rowset with data, you should only use:

- the **Fill** or **FillAppend** methods
- a record method (**SelectByKey**)
- **SQLExec**

If you specify a rowset object, you are creating a new rowset based on the structure of the specified rowset object, including any child rowsets. It will not contain any data. If you want to populate this type of rowset with data, use the **CopyTo** method or a SQL statement.

Note. You should *not* use the rowset **Select** or **SelectNew** methods for populating rowsets created using CreateRowset. Use **Fill** or **FillAppend** instead.

Restrictions on Using CreateRowset

The following methods and properties don't work with a rowset created using CreateRowset:

- **Select**
- **SelectNew**
- Any GUI methods (like **HideAllRows**)
- Any effective date methods or properties (like **EffDt**, **EffSeq**, or **GetCurrEffRow**)

In addition, rowsets created using CreateRowset are *not* automatically tied to the database. This means if you insert or delete rows, the rows will *not* be inserted or deleted in the database when you save the page.

Parameters

Parameter	Description
RECORD.*recname* | *&Rowset*	Specify either a record name or an existing rowset object.
FIELD.*fieldname*, **RECORD**.*recname* | *&Rowset*	Use **FIELD**.*fieldname*, **RECORD**.*recname* to specify a related display *record*. **FIELD**.*fieldname* refers to the controlling field, (not the related display field) while **RECORD**.*recname* refers to the related display record.
	If you specify *&rowset*, you are adding a child rowset object to the newly created rowset. This must be an existing rowset object.

Returns

An unpopulated, standalone rowset object.

Example

The following creates a simple rowset of just a single record per row:

```
&RS = CreateRowset(RECORD.QA_MYRECORD);
```

The following creates a rowset with the same structure as the specified rowset:

```
&RS2 = CreateRowset(&RS);
```

The following code creates a rowset structure composed of four records in an hierarchical structure, that is,

```
QA_INVEST_HDR
   QA_INVEST_LN
      QA_INVEST_TRANS
         QA_INVEST_DTL
```

Note that you have to start at the *bottom* of the hierarchy, and add the upper levels, not the other way around.

```
Local Rowset &RS, &RS2, &RS_FINAL;

&RS2 = CreateRowset(RECORD.QA_INVEST_DTL);
&RS = CreateRowset(RECORD.QA_INVEST_TRANS, &RS2);
&RS2 = CreateRowset(RECORD.QA_INVEST_LN, &RS);
&RS_FINAL = CreateRowset(RECORD.QA_INVEST_HDR, &RS2);
```

The following example reads all of the QA_MYRECORD records into a rowset, and returns the number of rows read:

```
&RS = CreateRowset(RECORD.QA_MYRECORD);
&NUM_READ = &RS.Fill();
```

To make a clone of an existing rowset, that is, to make two distinct copies, you can do the following:

```
&RS2 = CreateRowset(&RS);
&RS.CopyTo(&RS2);
```

The following code example is used for creating multiple children in a standalone rowset:

```
Local Rowset &rsBOCMRole, &rsBOCMRel, &rsBOCMUse;
```

```
&rsBOCMRole = CreateRowset(Record.BO_CM_ROLE);
&rsBOCMRel = CreateRowset(Record.BO_CM_REL);
&rsBOCMUse = CreateRowset(Record.BO_CM_USE);
&rsBOCM = CreateRowset(Record.BO_CM, &rsBOCMUse, &rsBOCMRole, &rsBOCMRel);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268, Chapter 1, PeopleCode Built-in Functions, GetLevel0 , page 253, Chapter 1, PeopleCode Built-in Functions, GetRecord , page 265, Chapter 1, PeopleCode Built-in Functions, GetField , page 242, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using Standalone Rowsets.

CreateSOAPDoc

Syntax

CreateSOAPDoc()

Description

Use the **CreateSOAPDoc** function to create an empty SOAPDoc object. Then use the SOAPDoc class methods and properties, as well as the XmlDoc class methods and properties to populate the SOAPDoc object.

Parameters

None.

Returns

A reference to a SOAPDoc object.

Example

```
Local SOAPDoc &MyDoc;

&MyDoc = CreateSOAPDoc();
```

See Also

Chapter 30, SOAPDoc Class , page 1587, Chapter 34, XmlDoc Class , page 1757.

CreateSQL

Syntax

CreateSQL([*sqlstring* [, *paramlist*]])

Where *paramlist* is an arbitrary-length list of values in the form:
inval1 [, *inval2*] ...

Description

The **CreateSQL** function instantiates a SQL object from the SQL class and opens it on the given *sqlstring* and input values. *sqlstring* is a PeopleCode string value giving the SQL statement.

Any errors in the SQL processing cause the PeopleCode program to be terminated with an error message.

You can use CreateSQL with no parameters to create an empty SQL object that can be used to assign properties before being populated and executed.

Opening and Processing sqlstring

If *sqlstring* is a SELECT statement, it is immediately bound with the *inval* input values and executed. The SQL object should subsequently be the subject of a series of **Fetch** method calls to retrieve the selected rows. If you want to fetch only a single row, use the SQLExec function instead. If you want to fetch a single row into a PeopleCode record object, use the record **Select** method.

If *sqlstring* is not a SELECT statement, and *either* there are some *inval* parameters, *or* there are no bind placeholders in the SQL statement, the statement is immediately bound and executed. This means that there is nothing further to be done with the SQL statement and the **IsOpen** property of the returned SQL object will be False. In this case, using the SQLExec function would generally be better. If you want to delete, insert or update a record object, use the record **Delete**, **Insert**, or **Update** methods.

If *sqlstring* is not a SELECT statement, there are no *inval* parameters, *and* there are bind placeholders in the SQL statement, the statement is neither bound nor executed. The resulting SQL object should subsequently be the subject of a series of **Execute** method calls to affect the desired rows.

Using Arrays with paramlist

You can use a parameter of type "Array of Any" in place of a list of bind values or in place of a list of fetch result variables. This is particularly useful when fetching an unknown number of results.

```
&Sql1 = CreateSql("Select * from " | &TableName);
&AAny = CreateArrayAny();

While &Sql1.Fetch(&AAny)
   /* Process the row in &AAny. */
   ...
End-While;
```

Because the Array of Any promotes to absorb any remaining select columns, it must be the last parameter for the SQL object Fetch method or (for results) SQLExec. For binding, it must be the only bind parameter, as it is expected to supply all the bind values needed.

Parameters

Parameter	Description
sqlstring	Specify a SQL string.
paramlist	Specify input values for the SQL string.

Returns

None.

Example

This SQL object should be used in a series of Fetch method calls:

```
Local SQL &SQL;
```

```
&SQL = CreateSQL("%SelectAll(:1) where EMPLID = :2", RECORD.ABSENCE_HIST, &EMPLID);
```

This SQL object has been opened, bound, and is already closed again:

```
&SQL = CreateSQL("Delete from %Table(:1) where EMPLID = :2", RECORD.ABSENCE_HIST,⇒
 &EMPLID);
```

This SQL object should be used in a series of Execute method calls:

```
&SQL = CreateSQL("Delete from %Table(:1) where EMPLID = :2");
```

This SQL object is created as an empty object in order to set properties before being executed:

```
&Sql = CreateSQL();
&Sql.Tracename = "SQL1";
&Sql.ReuseCursor = True;
&Sql.Open(......); /* do the deed */
```

See Also

Chapter 1, PeopleCode Built-in Functions, DeleteSQL , page 141, Chapter 1, PeopleCode Built-in Functions, FetchSQL , page 180, Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442, Chapter 1, PeopleCode Built-in Functions, StoreSQL , page 450, Chapter 31, SQL Class , page 1613, Chapter 31, SQL Class , Open, page 1626.

CreateXmlDoc

Syntax

CreateXmlDoc(*XmlString*)

Description

Use the **CreateXmlDoc** function to create an XmlDoc object. If you specify a Null string for *XmlString* (""), you create an empty XmlDoc object.

Considerations Using CreateXmlDoc

The following coding is either ignored or removed from the XmlDoc object that is created with this function:

- encoding attributes

  PeopleSoft only supports UFT-8 encoding. Any specified encoding statement is removed, as all XmlDoc objects are considered UFT-8.

- version attributes

  Regardless of what version is specified in *XmlString*, the version attribute in the generated XmlDoc object is 1.0.

Parameters

Parameter	Description
XmlString	Specify an XML string that you want to convert into an XmlDoc object that you can then manipulate using PeopleCode. You can also specify a Null string ("") to generate an empty XmlDoc object.

Returns

A reference to the newly created XmlDoc object.

Example

The following creates an empty XmlDoc object.

```
Local XmlDoc &MyDoc;

&MyDoc = CreateXmlDoc("");
```

See Also

Chapter 34, XmlDoc Class , page 1757.

CubicSpline

Syntax

CubicSpline(*DataPoints*, *Control_Option*, *Left_Constraint*, *Right_Constraint*)

Description

The **CubicSpline** function computes a cubic spline interpolation through a set of at least four datapoints.

Parameters

Parameter	Description
DataPoints	This parameter takes an array of array of number. The array's contents are an array of six numbers. The first two of these six numbers are the x and y points to be fit. The last four are the four coefficients to be returned from the function: **a**, **b**, **c** and **d**. **a** is the coefficient of the x^0 term, **b** is the coefficient of the x^1 term, **c** is the coefficient of the x^2 term, and **d** is the coefficient of the x^3 term.
Control_Option	Specifies the control option. This parameter takes either a number or constant value. The values are:

Numeric Value	Constant Value	Description
0	%SplineOpt_SlEstEst	Generate an internal estimate of the beginning and ending slope of the cubic piecewise equations
1	%SplineOpt_SlSetEst	Use the user-specified value for the slope of the leftmost point, and generate an estimate for the rightmost point

Numeric Value	Constant Value	Description
2	%SplineOpt_SlEstSet	Use the user-specified value for the slope of the rightmost point, and generate an estimate for the leftmost point
3	%SplineOpt_SlSetSet	Use the user-specified values for the slopes of the leftmost and rightmost points
4	%SplineOpt_CvEstEst	Generate an internal estimate of the beginning and ending curvature of the cubic piecewise equations
5	%SplineOpt_CvSetEst	Use the user-specified value for the curvature of the leftmost point, and generate an estimate for the rightmost point
6	%SplineOpt_CvEstSet	Use the user-specified value for the curvature of the rightmost point, and generate an estimate for the leftmost point
7	%SplineOpt_CvSetSet	Use the user-specified values for the curvatures of the leftmost and rightmost points
8	%SplineOpt_ClEstEst	Generate an internal estimate of the beginning and ending curl of the cubic piecewise equations
9	%SplineOpt_ClSetEst	Use the user-specified value for the curl of the leftmost point, and generate an estimate for the rightmost point
10	%SplineOpt_ClEstSet	Use the user-specified value for the curl of the rightmost point, and generate an estimate for the leftmost point
11	%SplineOpt_ClSetSet	Use the user-specified values for the curls of the leftmost and rightmost points
12	%SplineOpt_Natural	Generate a “natural” spline
13	%SplineOpt_ContCurl	Generate a spline wherein the equation for the first segment is the exact same equation for the second segment, and where the equation for the penultimate segment is the same as the equation for the last segment.

Parameter	Description
Left_Constraint	A single number for the constraint for the left point. Specify a zero if this parameter isn’t needed.
Right_Constraint	A single number for the constraint for the right point. Specify a zero if this parameter isn’t needed.

Returns

A modified array of array of numbers. The elements in the array correspond to the elements in the array used for *DataPoints*.

See Also

Chapter 1, PeopleCode Built-in Functions, HermiteCubic , page 285, Chapter 1, PeopleCode Built-in Functions, LinearInterp , page 314.

CurrEffDt

Syntax

CurrEffDt([*level_num*])

Description

CurrEffDt returns the effective date of the specified scroll level as a Date value.

Note. This function remains for backward compatibility only. Use the **EffDt** Rowset class property instead.

If no level is specified, **CurrEffDt**, returns the effective date of the current scroll level.

See Also

Chapter 27, Rowset Class , EffDt, page 1504, Chapter 27, Rowset Class , GetCurrEffRow, page 1491, Chapter 27, Rowset Class , EffSeq, page 1505, *PeopleCode Developer's Guide*, Data Buffer Access.

Returns

Returns a Date value equal to the current effective date of the specified scroll level.

Example

```
If INSTALLATION.POSITION_MGMT = "P" Then
   If All(POSITION_NBR) Then
      If (EFFDT = CurrEffDt(1) and
            EFFSEQ >= CurrEffSeq(1)) or
            (EFFDT > CurrEffDt(1) and
               EFFDT = %Date) Then
         Gray_employment( );
       End-if;
   End-if;
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CurrEffRowNum , page 114, Chapter 1, PeopleCode Built-in Functions, CurrEffSeq , page 115, Chapter 1, PeopleCode Built-in Functions, CurrentLevelNumber , page 116.

CurrEffRowNum

Syntax

CurrEffRowNum([*level_num*])

Description

CurrEffRowNum returns the effective row number of the selected scroll level.

Note. This function remains for backward compatibility only. Use the **RowNumber** Row class property, in combination with the **GetCurrEffRow** Rowset method instead.

If no level is specified, it returns the effective row number of the current level.

See Also

Chapter 26, Row Class , RowNumber, page 1476, Chapter 27, Rowset Class , GetCurrEffRow, page 1491, *PeopleCode Developer's Guide*, Data Buffer Access.

Example

```
&ROW = CurrEffRowNum(1);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CurrEffSeq , page 115, Chapter 1, PeopleCode Built-in Functions, CurrentLevelNumber , page 116, Chapter 1, PeopleCode Built-in Functions, CurrEffRowNum , page 114.

CurrEffSeq

Syntax

CurrEffSeq([*level_num*])

Description

CurrEffSeq determines the effective sequence of a specific scroll area.

Note. This function remains for backward compatibility only. Use the **EffSeq** rowset class property instead.

If no level is specified, **CurrEffSeq** returns the effective sequence of the current scroll level.

See Also

Chapter 27, Rowset Class , EffSeq, page 1505, Chapter 27, Rowset Class , GetCurrEffRow, page 1491, Chapter 27, Rowset Class , DeleteEnabled, page 1504, *PeopleCode Developer's Guide*, Data Buffer Access.

Returns

Returns a Number representing the effective sequence of the specified scroll level.

Example

```
If INSTALLATION.POSITION_MGMT = "P" Then
   If All(POSITION_NBR) Then
      If (EFFDT = CurrEffDt(1) and
            EFFSEQ >= CurrEffSeq(1)) or
            (EFFDT > CurrEffDt(1) and
               EFFDT = %Date) Then
         Gray_employment( );
       End-if;
   End-if;
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CurrEffDt , page 114, Chapter 1, PeopleCode Built-in Functions, CurrentLevelNumber , page 116, Chapter 1, PeopleCode Built-in Functions, CurrEffRowNum , page 114.

CurrentLevelNumber

Syntax

```
CurrentLevelNumber()
```

Description

The **CurrentLevelNumber** function returns the scroll level where the function call is located.

Returns

Returns a Number value equal to the scroll level where the function is being called. The function returns 0 if the field where the function is called is not in a scroll area.

Example

```
&LEVEL = CurrentLevelNumber();
```

See Also

Chapter 1, PeopleCode Built-in Functions, CurrentRowNumber , page 116, Chapter 1, PeopleCode Built-in Functions, FetchValue , page 181.

CurrentRowNumber

Syntax

```
CurrentRowNumber([level])
```

Description

Use **CurrentRowNumber** to determine the row number of the row currently displayed in a specific scroll area.

Note. This function remains for backward compatibility only. Use the **RowNumber** Row property instead.

This function can determine the current row number on the level where the function call resides, or on a higher scroll level. It won't work on a scroll level below the one where the PeopleCode program resides.

See Also

Chapter 1, PeopleCode Built-in Functions, GetRow , page 267, Chapter 26, Row Class , RowNumber, page 1476, *PeopleCode Developer's Guide,* Data Buffer Access.

Parameters

Parameter	Description
level	A Number specifying the scroll level from which the function returns the current row number. If the level parameter is omitted, it defaults to the scroll level where the function call resides.

Returns

Returns a Number value equal to the current row number on the specified scroll level. The current number is the row where the PeopleCode program is being processed, or, if level specifies a higher level scroll, CurrentRowNumber returns the row number of the parent or grandparent row.

Example

CurrentRowNumber is typically used in component buffer functions to return the current row of the parent scroll of the target:

```
&VAL = FetchValue(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(), ⇒
BUS_EXPENSE_DTL.CHARGE_DT, &COUNT);
```

The following example checks if the current row number is equal to the active row count (that is, whether the active row is the last record on the scroll):

```
If CurrentRowNumber() = ActiveRowCount(EMPLID) Then
   det_employment_dt();
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, ActiveRowCount , page 29, Chapter 1, PeopleCode Built-in Functions, CurrentLevelNumber , page 116, Chapter 1, PeopleCode Built-in Functions, FetchValue , page 181.

Date

Syntax

Date(*date_num*)

Description

Date takes a number in the form YYYYMMDD and returns a corresponding Date value. If the date is invalid, Date displays an error message.

Warning! Make sure that you pass a four-digit year in the year parameter of this function. Two-digit values are interpreted literally: 93, for example, represents the year 93 AD.

Returns

Returns a date equal to the date specified in *date_num*.

Example

Set the temporary variable &HIREDate to a date field containing the date July 1, 1997:

```
&HIREDate = Date(19970701);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date3 , page 118, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Day , page 127, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129, Chapter 1, PeopleCode Built-in Functions, Month , page 327, Chapter 1, PeopleCode Built-in Functions, Weekday , page 501, Chapter 1, PeopleCode Built-in Functions, Year , page 509.

Date3

Syntax

Date3(*year*, *month*, *day*)

Description

The **Date3** function accepts a date expressed as three integers: *year*, *month*, and *day*. It returns a corresponding Date value. If the date is invalid, the Date3 displays an error message.

Warning! Make sure that you pass a four-digit year in the year parameter of this function. Two-digit values will be interpreted literally: 93, for example, represents the year 93 AD.

Parameters

Parameter	Description
year	An integer for the year in the form YYYY.
month	An integer from 1 to 12 designating the month.
day	An integer from 1 to 31 designating the day of the month.

Returns

Returns a Date value equal to the date specified in the function parameters.

Example

The following PeopleCode Date3 function returns the first day of the year in which the employee was hired:

```
Date3(HIRE_YEAR, 1, 1);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date , page 117, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Day , page 127, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129.

DatePart

Syntax

DatePart(*datetime_value*)

Description

Use **DatePart** to determine a date based on a provided DateTime value.

Returns

Returns a Date value equal to the date part of a specified DateTime value.

Example

The following statement sets &D2 to a Date value for 11/12/97:

```
&D1 = DateTimeValue("11/12/97 10:23:15 AM");
&D2 = DatePart(&D1);
```

DateTime6

Syntax

DateTime6(*year*, *month*, *day*, *hour*, *minute*, *second*)

Description

DateTime6 returns a DateTime value based on integer values for the *year*, *month*, *day*, *hour*, *minute*, and *second*. If the result of this function is not an actual date, there is a runtime error.

Warning! Make sure that you pass a four-digit year in the year parameter of this function. Two-digit values will be interpreted literally: 93, for example, represents the year 93 AD.

Parameters

Parameter	Description
year	A four-digit number representing the year.
month	A number between 1 and 12 representing the month.
day	A number representing the day of the month.
hour	A number from 0 to 23 representing the hour of the day.
minute	A number from 0 to 59 representing the minute of the hour.
second	A number from 0 to 59.999999 representing seconds.

Returns

Returns a DateTime value based on the integers provided.

Example

The following example sets &DT to a DateTime value equal to 10:09:20 on March 15, 1997:

```
&DT = DateTime6(1997, 3, 15, 10, 9, 20);
```

DateTimeToHTTP

Syntax

DateTimeToHTTP(*datetime*)

Description

The **DateTimeToHTTP** function converts any datetime value to a datetime string in the format specified by HTTP 1.0 and 1.1 standards.

Note. Because the HTTP protocol is used to interchange information between diverse computing systems, the value returned from this function is always the ”US English” form of weekdays and months. If you want the value to use the localized form, use the **DateTimeToLocalizeString** function instead.

The standard HTTP datetime has the following fixed length format:

```
<dow><,><sp><dd><sp><mon><sp><year><sp><hh><:><mm><:><ss><sp><GMT>
```

where:

Value	Description
<dow>	a 3-character day of week name, one of Sun, Mon, Tue, Wed, Thu, Fri, Sat. This is the localized version of this name.
<,>	a literal comma character
<sp>	a literal space character
<dd>	a 2-digit day of month, such as 02 or 22.
<mon>	is a 3-character month name, one of Jan, Feb, Mar, and so on. This is the localized version of this name.
<year>	a 4-digit year number
<hh>	a 24-hour hour number, such as 00 or 13
<mm>	a 2-digit minute number, such as 01 or 56
<ss>	a 2-digit second number, such as 03 or 59
<GMT>	a literal 3-character GMT.

As indicated by the trailing GMT, this datetime format is always expressed in GMT (or UTC, which is declared to be the same for the purposes of HTTP).

Parameters

Parameter	Description
datetime	Specify the datetime value you want converted to HTTP format. This datetime is assumed to be in the base timezone of the installation.

Returns

A string containing the converted datetime.

Example

```
&gmtdate = DateTimeToHTTP(AddToDateTime(%DateTime, 0,0,0,0,600,0));

%Response.setHeader("Last-Modified", &gmtdate);
```

See Also

Chapter 1, PeopleCode Built-in Functions, AddToDateTime , page 40, Chapter 1, PeopleCode Built-in Functions, DateTimeToLocalizedString , page 121, Chapter 1, PeopleCode Built-in Functions, FormatDateTime , page 192.

DateTimeToLocalizedString

Syntax

DateTimeToLocalizedString({*datetime* | *date*}, [*Pattern*])

Description

The **DateTimeToLocalizedString** function converts either *datetime* or *date* to a localized string. You can also specify a particular pattern to convert *datetime* or *date* to.

The *Pattern* is optional. Only specify *Pattern* if necessary.

If you need to change the pattern for each language, change the first message in Message Catalog set number 138. This is a format for each language.

Parameters

Parameter	Description
Datetime | *Date*	Specify either the datetime or date that you want to convert.
Pattern	Specify the pattern you want to the localized datetime or date to be converted to.

Using the Pattern Parameter

Pattern takes a string value, and indicates how you want the date or datetime converted.

The valid values for *Pattern* are as follows.

Note. The values for pattern are case-sensitive. For example, if you specify a lowercase m, you get minutes, while an uppercase M displays the month.

Symbol	Definition	Type	Example
G	Era designator	Text	AD
y	Year	Number	1996

Symbol	Definition	Type	Example
M	Month in year	Text&Number	July&07
d	Day in month	Number	10
h	Hour in am/pm	Number (1-12)	12
H	Hour in day	Number (0-23)	0
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
E	Day in week	Text	Tuesday
a	am/pm marker	Text	PM
k	Hour in day	Number (1-24)	24
K	Hour in am/pm	Number (0-11)	0
’	Escape for text	Delimiter	
’’	Single quote	Literal	’

The number of pattern letters determine the format.

Type	Pattern Format
Text	If 4 or more pattern letters are used, the full form is used. If less than 4 pattern letters are used, the short or abbreviated form is used if one exists.

Type	Pattern Format
Number	Use the minimum number of digits. Shorter numbers are zero-padded to this amount. The year is handled specially; that is, if the count of 'y' is 2, the year is truncated to 2 digits.
Text&Number	If 3 or more pattern letters are used, text is used, otherwise, a number is used.

Any characters in *Pattern* are not in the ranges of ['a'..'z'] and ['A'..'Z'] are treated as quoted text. For instance, characters like ':', '.', ' ', '#' and '@' appear in the resulting string even they're not within single quotes.

A pattern containing any invalid pattern letter results in a runtime error.

Examples using a United Stated locale:

Pattern	Result
"yyyy.MM.dd G 'at' hh:mm:ss"	1996.07.10 AD at 15:08:56
"EEE, MMM d, ''yy"	Wed, July 10, '96
"h:mm a"	12:08 PM
"hh 'o''clock' a"	12 o'clock PM
"K:mm a"	0:00 PM
"yyyyy.MMMMM.dd GGG hh:mm aaa"	1996.July.10 AD 12:08 PM

Returns

A string.

Example

```
REM****************************************************************;
Function ConvertDateToDTTM(&Date As date) Returns datetime ;
REM ************************************************************;
   &String = DateTimeToLocalizedString(&Date, "M/d/y");
   &String = &String | " 00:00:00.000000";
   &DateTime = DateTimeValue(&String);
   Return &DateTime;
End-Function;
```

See Also

Chapter 1, PeopleCode Built-in Functions, FormatDateTime , page 192, Chapter 1, PeopleCode Built-in Functions, DateTimeToHTTP , page 120.

DateTimeToTimeZone

Syntax

DateTimeToTimeZone(*OldDateTime, SourceTimeZone, DestinationTimeZone*)

Description

DateTimeToTimeZone converts datetimes from the datetime specified by *SourceTimeZone* to the datetime specified by *DestinationTimeZone*.

Considerations Using this Function

Typically, this function is used in PeopleCode, *not* for displaying time. If you take a datetime value, convert it from base time to client time, then try to display this time, depending on the user settings, when the time is displayed the system might try to do a *second* conversion on an already converted datetime. This function could be used as follows: suppose a user wanted to check to make sure a time was in a range of times on a certain day, in a certain timezone. If the times were between 12 AM and 12 PM in EST, these resolve to 9 PM and 9 AM PST, respectively. The start value is *after* the end value, which makes it difficult to make a comparison. This function could be used to do the conversion for the comparison, in temporary fields, and not displayed at all.

Parameters

Parameter	Description
OldDateTime	Specify the datetime value to be converted.
SourceTimeZone	Specify the time zone that *OldDateTime* is in. Valid values are: • *timezone* - a time zone abbreviation or a field reference to be used for converting *OldDateTime*. • **Local** - use the local time zone for converting *OldDateTime*. • **Base** - use the base time zone for converting *OldDateTime*.
DestinationTimeZone	Specify the time zone that you want to convert *OldDateTime* to. Valid values are: • *timezone* - a time zone abbreviation or a field reference to be used for converting *OldDateTime*. • **Local** - use the local time zone for converting *OldDateTime*. • **Base** - use the base time zone for converting *OldDateTime*.

Returns

A converted datetime value.

Example

The following example. TESTDTTM, is a datetime field with a value 01/01/99 10:00:00. This example converts TESTDTTM from Pacific standard time (PST) to eastern standard time (EST).

```
&NEWDATETIME = DateTimeToTimeZone(TESTDTTM, "PST", "EST");
```

&NEWDATETIME will have the value 01/01/99 13:00:00 because EST is three hours ahead of PST on 01/01/99, so three hours are added to the datetime value.

See Also

Chapter 1, PeopleCode Built-in Functions, ConvertDatetimeToBase , page 85, Chapter 1, PeopleCode Built-in Functions, ConvertTimeToBase , page 87, Chapter 1, PeopleCode Built-in Functions, FormatDateTime , page 192, Chapter 1, PeopleCode Built-in Functions, IsDaylightSavings , page 302, Chapter 1, PeopleCode Built-in Functions, TimeToTimeZone , page 460, Chapter 1, PeopleCode Built-in Functions, TimeZoneOffset , page 461, *PeopleSoft Server Tools Administration*, PeopleTools Utilities.

DateTimeValue

Syntax

DateTimeValue(*string*)

where *string* has the format:

MM/DD/YY[*YY*] *hh*:*mm*:*ss.ssssss* [{**AM**|**PM**}]

or

MM.DD.YY[*YY*] *hh*:*mm*:*ss.ssssss* [{**AM**|**PM**}]

Description

Use **DateTimeValue** to derive a DateTime value from a string representing a date and time.

If the AM | PM part is omitted, the value is assumed to be based on a 24-hour clock. This function is language-sensitive. However, be aware that it returns date values based on Windows settings, not based on PeopleSoft language preference settings. It is therefore unaffected by calls to **SetLanguage**.

If the Windows Regional Settings Date properties are set to DD/MM/YY, then

```
&DTM = DateTimeValue("10/09/97 10:34:36");
```

will return a Datetime value equal to September 10, 1997, 10:34:36.

If the Windows Regional Settings Date properties are set to MM/DD/YY, then the same function call will return a value equal to October 9, 1997, 10:34:36.

Using this Function in Fields Without a Default Century Setting

This function may derive the wrong century setting if passed a 2-character year *and* DateTimeValue is executing in a PeopleCode event not associated with a field that has a default century setting.

For example, assume that TEST_DATE is a date field with a default century setting of 10. TEST_FIELD is a field with *no* default century setting. If the following PeopleCode program is executing in TEST_FIELD, the date will be calculated incorrectly:

```
TEST_DATE = DateTimeValue("10/13/11 15:34:25");
```

Though TEST_DATE has a century setting, it isn't used because the PeopleCode fired in TEST_FIELD. Instead, the code uses the 50/50 rule and calculates the year to be 2011 (instead of 1911).

Returns

Returns a DateTime value equal to the datetime specified in the input string.

Example

Both of the following examples set &Date_TIME to a Datetime value equal to 10/13/97 10:34:25 PM. These examples assume that the Windows Regional Date properties are set to the US English defaults:

```
&Date_TIME = DateTimeValue("10/13/97 10:34:25 PM");

&Date_TIME = DateTimeValue("10/13/97 22:34:25");
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date , page 117, Chapter 1, PeopleCode Built-in Functions, Date3 , page 118, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Day , page 127, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129, Chapter 1, PeopleCode Built-in Functions, Month , page 327, Chapter 1, PeopleCode Built-in Functions, Weekday , page 501, Chapter 1, PeopleCode Built-in Functions, Year , page 509.

DateValue

Syntax

DateValue(*date_str*)

Description

DateValue converts a date string and returns the result as a Date type. *date_str* must be a string in the active date format, based on the Windows Regional Date property settings.

Be aware that it returns date values based on Windows settings, not based on PeopleSoft language preference settings. It is therefore unaffected by calls to **SetLanguage**.

If the Windows Regional Settings Date properties are set to DD.MM.YY, then

```
&DTM = DateValue("10/09/97");
```

returns a Datetime value equal to September 10, 1997.

If the Windows Regional Settings Date properties are set to MM/DD/YY, then the same function call returns a value equal to October 9, 1997.

Using this Function in Fields Without a Default Century Setting

This function may derive the wrong century setting if passed a 2-character year *and* DateValue is executing in a PeopleCode event not associated with a field that has a default century setting.

For example, assume that TEST_DATE is a date field with a default century setting of 10. TEST_FIELD is a field with *no* default century setting. If the following PeopleCode program is executing in TEST_FIELD, the date will be calculated incorrectly:

```
TEST_DATE = DateValue("10/13/11");
```

Though TEST_DATE has a century setting, it isn't used because the PeopleCode fired in TEST_FIELD. Instead, the code uses the 50/50 rule and calculates the year to be 2011 (instead of 1911).

Returns

Returns a Date value equal to the date specified in *date_str*.

Example

If the Windows client is set up with standard US date settings, the following DateValue function call converts the date string to a Date value equal to October 24, 1997:

```
&DT = DateValue("10/24/97");
```

If a workstation is set up with the typical European date settings, the following DateValue function converts the date string to the equivalent Date value:

```
&DT = DateValue("24.10.97");
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date , page 117, Chapter 1, PeopleCode Built-in Functions, Date3 , page 118, Chapter 1, PeopleCode Built-in Functions, DateTimeValue , page 125, Chapter 1, PeopleCode Built-in Functions, Day , page 127, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129, Chapter 1, PeopleCode Built-in Functions, Month , page 327, Chapter 1, PeopleCode Built-in Functions, Weekday , page 501, Chapter 1, PeopleCode Built-in Functions, Year , page 509.

Day

Syntax

Day(*dt_val*)

Description

The **Day** function determines an integer representing the day of the month based on a Date or DateTime value.

Returns

Returns a Number value equal to the day of the month for *dt_val*. The return value is an integer from 1 to 31.

Example

If HIRE_DATE is November, 1, 1997, the following Day function returns the integer 1:

```
&FIRST_DAy = Day(HIRE_DATE);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date , page 117, Chapter 1, PeopleCode Built-in Functions, Date3 , page 118, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Days , page 128, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129, Chapter 1, PeopleCode Built-in Functions, Month , page 327, Chapter 1, PeopleCode Built-in Functions, Weekday , page 501, Chapter 1, PeopleCode Built-in Functions, Year , page 509.

Days

Syntax

Days(*dt_val*)

Description

The **Days** function returns the Julian date format for the *dt_val* specified. This function accepts a Date, DateTime, or Time value parameter.

Returns

Returns a Number value equal to the Julian date format for *dt_val*.

Example

To find the number of days between two dates, use the Days function on both dates, and subtract one from the other:

```
&NUM_DAYS = Abs(Days(HIRE_Date) - Days(RELEASE_Date));
```

See Also

Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129, Chapter 1, PeopleCode Built-in Functions, Month , page 327, Chapter 1, PeopleCode Built-in Functions, Weekday , page 501, Chapter 1, PeopleCode Built-in Functions, Year , page 509.

Days360

Syntax

Days360(*date_val1, date_val2*)

Description

The **Days360** function returns the number of days between the Date values *date_val1* and *date_val2* using a 360-day year (twelve 30-day months). Use this function to help compute payments if your accounting system is based on twelve 30-day months.

If *date_val2* occurs before *date_val1*, Days360 returns a negative number.

Example

The following example sets &NUMDAYS to the number of days between TERM_START_DT and PMT_DT based on a 360-day calendar:

```
&NUMDAYS = Days360(TERM_START_DT, PMT_DT);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date , page 117, Chapter 1, PeopleCode Built-in Functions, Date3 , page 118, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Day , page 127, Chapter 1, PeopleCode Built-in Functions, Days , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129, Chapter 1, PeopleCode Built-in Functions, Month , page 327, Chapter 1, PeopleCode Built-in Functions, Weekday , page 501, Chapter 1, PeopleCode Built-in Functions, Year , page 509.

Days365

Syntax

Days365(*date_val1, date_val2*)

Description

Returns the number of days between the Date values *date_val1* and *date_val2* using a 365-day year. Use this function to help compute payments if your accounting system is based on a 365-day year.

If *date_val2* occurs before *date_val1*, Days365 returns a negative number.

Returns

Returns a Number value equal to the number of days between the two dates in a 365-day calendar.

Example

The following example sets &NUMDAYS to the number of days between and TERM_ START_DT and PMT_DT, based on a 365-day calendar:

```
&NUMDAYS = Days360(TERM_START_DT, PMT_DT);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date , page 117, Chapter 1, PeopleCode Built-in Functions, Date3 , page 118, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Day , page 127, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Month , page 327, Chapter 1, PeopleCode Built-in Functions, Weekday , page 501, Chapter 1, PeopleCode Built-in Functions, Year , page 509.

DBCSTrim

Syntax

DBCSTrim(*source_str*)

Description

DBCSTrim removes a trailing DBCS lead byte at the end of the string. Use this function to clean up a DBCS string after a byte-oriented operation (such as **Substringb**) that may have split a DBCS character at the end of the string.

Returns

Returns a String formed by trimming the trailing DBCS lead byte. If there is no trailing DBCS lead byte, the string returned is identical to the source string.

See Also

Chapter 1, PeopleCode Built-in Functions, Substringb , page 453.

DBPatternMatch

Syntax

DBPatternMatch(*Value, Pattern, CaseSensitive*)

Description

Use the **DBPatternMatch** function to match the string in *Value* to the given pattern.

You can use wildcard characters % and _ when searching. % means find all characters, while _ means find a single character. For example, if you wanted to find if the string in *Value* started with the letter M, you'd use "M%" for *Pattern*. If you wanted to find either DATE or DATA, use "DAT_" for *Pattern*.

These characters can be escaped (that is, ignored) using a \. For example, if you want to search for a value that contains the character %, use \% in *Pattern*.

If *Pattern* is an empty string, this function retrieves the value just based on the specified case-sensitivity (that is, specifying "" for *Pattern* is the same as specifying "%").

Parameters

Parameter	Description
Value	Specify the string to be searched.
Pattern	Specify the pattern to be used when searching.
CaseSensitive	Specify whether the search is case-sensitive. This parameter takes a Boolean value: True, the search is case-sensitive, False, it is not.

Returns

Returns a Boolean value. True if the string matches the pattern, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, Find , page 185, Chapter 1, PeopleCode Built-in Functions, Findb , page 186.

Declare Function

Syntax

PeopleCode Function Syntax

```
Declare Function function_name PeopleCode record_name.field_name event_type
```

External Function Syntax

```
Declare Function function_name Library lib_name
   [ALIAS module_name ]
   [paramlist]
   [RETURNS ext_return_type [As pc_type]]
```

Where *paramlist* is:

```
([ext_param1 [, ext_param2] . . .)
```

and where *ext_paramn* is:

```
ext_datatype [{REF|VALUE}] [As pc_return_type]
```

Description

PeopleCode can call PeopleCode functions defined in any field on any record definition. You can create special record definitions whose sole purpose is to serve as function libraries. By convention, PeopleCode functions are stored in FieldFormula PeopleCode, in record definitions with names beginning in FUNCLIB_.

PeopleCode can also call external programs that reside in dynamic link libraries. You must declare either of these types of functions at the top of the calling program using the **Declare Function** statement.

To support processes running on an application server, you can declare and call functions compiled in dynamic link libraries on windows (*.DLL files) and shared libraries on UNIX (lib*.so files.) The PeopleCode declaration and function call syntax is the same regardless of platform, but UNIX libraries must be compiled with an interface function.

See *PeopleCode Developer's Guide*, PeopleCode and PeopleSoft Internet Architecture, Calling Dynamic Link Library Functions on the Application Server.

PeopleCode Functions

You can call a PeopleCode function defined on any record definition, provided you declare it at the top of the calling program. The declaration identifies the function name, as well as the record, field, and event type where the function definition resides. The function parameters and return type are not declared; they are determined from the function definition.

Note. You can define functions only in Record Field PeopleCode. You can't define functions in Component PeopleCode, Component Record Field PeopleCode, and so on.

External Library Functions

Function declarations define routines in an external (C-callable) library. The function declaration provides the name of the library, an optional alias *module_name*, a list of parameters to pass to the function, an optional **Returns** clause specifying the type of any value returned by the external function, and the PeopleCode data type into which to convert the returned value. The library must be a DLL accessible by Windows or a shared library accessible by UNIX.

After you have declared an external library function, you can call it the same way as an external PeopleCode function. Like PeopleCode functions, you must pass the number of parameters the library function expects. Calls to external functions suspend processing: this means that you should exercise caution to avoid "think-time" errors when calling the function in the following PeopleCode events:

- SavePreChange.
- SavePostChange.
- Workflow.
- RowSelect.
- Any PeopleCode event that fires as a result of a ScrollSelect (or one of its relatives) function calls, or a Select (or one of its relatives) Rowset class method.

See *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Think-Time Functions.

Parameters

The following are the parameters for the PeopleCode Function syntax:

Parameter	Description
function_name	Name of the function.
PeopleCode	Reserved word that identifies the function as a PeopleCode function.
recordname.fieldname	Specifies the record and field where the function is located.
event_type	Component Processor event with which the function is associated.

Note. *event_type* can be used to specify *record field* events only. You can't specify a component record field event, a component record event, and so on.

The following are the parameters for the External Function syntax:

Parameter	Description
function_name	Name of the function.
Library	Reserved word that identifies the function as an external library function.
lib_name	A string representing the name of the external library. The external routine must be located in a DLL named *lib_name* accessible by Windows, or an equivalent shared library in a UNIX system.
Alias *module_name*	Optionally specifies the name of the function's entry point within the shared library. This is needed only if the C function name differs from *function_name* in the PeopleCode external function declaration. The external module is invoked using the __stdcall calling convention on Windows.
Paramlist	List of parameters expected by the function, each in the form:
	ext_datatype [{**Ref**|**Value**}] [**As** *pc_type*]
ext_type	The datatype of the parameter expected by the function. To specify the type you can use any of the following: • BOOLEAN • INTEGER • LONG • UINTEGER • ULONG • STRING • STRING • FLOAT • DOUBLE
Ref | **Value**	Optionally use one of these two reserved words to specify whether the parameter is passed by reference or by value. If **Ref** is specified, it is passed by pushing a reference (pointer) on the stack. If **Value** is specified the value is pushed on the stack (for integers, and so on.) If neither is specified, **Ref** is assumed.
AS *pc_type*	Specifies PeopleCode data type of the value passed to the function. You can choose between PeopleCode data types String, Number, Date, Boolean, and Any.

Parameter	Description
Returns *ext_return_type*	Specifies the data type of any value returned by the function. The **Returns** clause is omitted if the function is void (returns no value.) To specify the return type you can use any of the following: • BOOLEAN • INTEGER • LONG • UINTEGER • ULONG • FLOAT • DOUBLE
	The types String and LString are not allowed for the result type of a function.
As *pc_return_type*	Specifies the PeopleCode data type of the variable or field into which to read the returned value. You can choose between PeopleCode data types String, Number, Date, Boolean, and Any. If the **As** clause is omitted, PeopleTools selects an appropriate type based on the type of value returned by the external function (for example, all integer and floating point types are converted to Number).

Example

Assume you have defined a PeopleCode function called **VerifyZip**. The function definition is located in the record definition FUNCLIB_MYUTILS, in the record field ZIP_EDITS, attached to the FieldFormula event. You would declare the function using the following statement:

```
declare function verifyzip PeopleCode FUNCLIB_MYUTILS.ZIP_EDITS FieldFormula;
```

Now assume you want to declare a function called **PCTest**, in PSUSER.DLL. It takes an integer and returns an integer. You would write this declare statement:

```
declare function pctest library "psuser.dll" (integer value as number) returns⇒
 integer as number;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Function , page 196.

Decrypt

Syntax

Decrypt(*KeyString*, *EncryptedString*)

Description

The **Decrypt** function decrypts a string previously encrypted with the **Encrypt** function. This function is generally used with merchant passwords. For this function to decrypt a string successfully, you must use the same *KeyString* value used to encrypt the string.

Parameters

Parameter	Description
KeyString	Specify the key used for encrypting the string. You can specify a NULL value for this parameter, that is, two quotation marks with no blank space between them ("").
EncryptedString	Specify the string you want decrypted.

Returns

A clear text string.

Example

Encrypt and Decrypt support only strings.

```
&AUTHPARMS.WRKTOKEN.Value = Decrypt("", RTrim(LTrim(&MERCHANTID_⇒
REC.CMPAUTHNTCTNTOKEN.Value)));
```

See Also

Chapter 1, PeopleCode Built-in Functions, Encrypt , page 159, Chapter 1, PeopleCode Built-in Functions, Hash , page 284.
PeopleTools Security PeopleBook

Degrees

Syntax

Degrees(*angle*)

Description

Degrees converts the given angle from radian measurement to degree measurement.

Parameters

Parameter	Description
angle	The size of an angle in radians.

Returns

The size of the given angle in degrees.

Example

The following example returns the equivalent size in degrees of an angle measuring 1.2 radians:

```
&DEGREE_SIZE = Degrees(1.2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Acos , page 29, Chapter 1, PeopleCode Built-in Functions, Asin , page 45, Chapter 1, PeopleCode Built-in Functions, Atan , page 46, Chapter 1, PeopleCode Built-in Functions, Cos , page 91, Chapter 1, PeopleCode Built-in Functions, Cot , page 92, Chapter 1, PeopleCode Built-in Functions, Radians , page 356, Chapter 1, PeopleCode Built-in Functions, Sin , page 439, Chapter 1, PeopleCode Built-in Functions, Tan , page 456.

DeleteAttachment

Syntax

DeleteAttachment(*URLSource, DirAndFileName*)

where *URLSource* can have *one* of the following forms:
URL.*URLname*

Or a string URL, such as
ftp://user:password@ftp.ps.com/

Description

The **DeleteAttachment** function enables you to delete a file on a specified system.

DeleteAttachment does *not* generate any type of “Are you sure” message. If you want the end-user to verify the deletion before it’s performed, you must write your own checking code in your application.

File Name Considerations

When the file is transferred, the following characters are replaced with an underscore:

- space
- semicolon
- plus sign
- percent sign
- ampersand
- apostrophe
- exclamation point
- pound sign
- dollar sign

Parameters

Parameter	Description
URLSource	A reference to a URL. This can be either a URL name, in the form URL.*URLName*, or a string. This is where the file will end up.
DirAndFileName	A directory and filename. This is appended to the *URLSource* to make up the full URL of the file.

Note. The *URLSource* requires "/" slashes. Because the *DirAndFileName* parameter is appended to the URL, it also requires only "/" slashes. "\" are *not* supported in anyway for either the *URLSource* the *DirAndFileName* parameter.

Returns

An integer value. You can check for either the integer or the constant:

Numeric Value	Constant Value	Description
0	%Attachment_Success	File was attached successfully.
1	%Attachment_Failed	File was not successfully attached.
2	%Attachment_Cancelled	File attachment didn't complete because the operation was canceled by the user.
7	%Attachment_DestSystNotFound	Cannot locate destination system for ftp.
8	%Attachment_DestSystFailedLogin	Unable to login to destination system for ftp.
9	%Attachment_FileNotFound	Cannot locate file.
10	%Attachment_DeleteFailed	Cannot delete file.

Example

```
&retcode = DeleteAttachment(URL.BKFTP, ATTACHSYSFILENAME);

If (&retcode = %Attachment_Success) Then
   MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment succeeded");
   UnHide(ATTACHADD);
   Hide(ATTACHVIEW);
   Hide(ATTACHDELETE);
   ATTACHUSERFILE = "";
   ATTACHSYSFILENAME = "";
End-If;

If (&retcode = %Attachment_Failed) Then
   MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed");
End-If;
If (&retcode = %Attachment_Cancelled) Then
   MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment cancelled");
End-If;
If (&retcode = %Attachment_FileTransferFailed) Then
   MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed: File⇒
 Transfer did not succeed");
End-If;

/* following error message only in PeopleSoft Internet Architecture */

If (&retcode = %Attachment_NoDiskSpaceAppServ) Then
   MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed: No disk⇒
```

```
 space on the app server");
End-If;

/* following error message only in PeopleSoft Internet Architecture */

If (&retcode = %Attachment_NoDiskSpaceWebServ) Then
   MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed: No disk⇒
 space on the web server");
End-If;

If (&retcode = %Attachment_FileExceedsMaxSize) Then
   MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed: File⇒
 exceeds the max size");
End-If;
If (&retcode = %Attachment_DestSystNotFound) Then
   MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed: Cannot⇒
 locate destination system for ftp");
End-If;
If (&retcode = %Attachment_DestSysFailedLogin) Then
   MessageBox(0, "File Attachment Status", 0, 0, "DeleteAttachment failed: Unable⇒
 to login into destination system for ftp");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, AddAttachment , page 31, Chapter 1, PeopleCode Built-in Functions, GetAttachment , page 234, Chapter 1, PeopleCode Built-in Functions, PutAttachment , page 353, Chapter 1, PeopleCode Built-in Functions, ViewAttachment , page 494, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Understanding File Attachment Architecture.

DeleteEmailAddress

Syntax

DeleteEmailAddress(*Type*)

Description

Use the **DeleteEmailAddress** function to delete the email address associated with the specified type for the current user. You can only have *one* email address of a specific type for a user.

Note. You can only delete the Primary Email Address if it's the only address. If the email address you want to delete is marked as the primary email address, and it is *not* the only email address, you must mark another email address as primary before you can delete the email address you want to delete. Use the MarkPrimaryEmailAddress function to set the primary email address.

Parameters

Parameter	Description
Type	Specify the type that you want to change the email address to. This parameter takes a string value. The valid values are:

Value	Description
BB	Blackberry email address
BUS	Business email address
HOME	Home email address
OTH	Other email address
WORK	Work email address

Returns

None.

See Also

Chapter 1, PeopleCode Built-in Functions, AddEmailAddress , page 35, Chapter 1, PeopleCode Built-in Functions, ChangeEmailAddress , page 63, Chapter 1, PeopleCode Built-in Functions, MarkPrimaryEmailAddress , page 317.

DeleteImage

Syntax

```
DeleteImage(scrollpath, target_row, [recordname.]fieldname)
```

where *scrollpath* is:

```
[SCROLL.level1_recname, level1_row, [SCROLL.level2_recname, level2_row,]]
SCROLL.target_recname
```

Description

The **DeleteImage** function enables a user to remove an image associated with a record field.

Note. To update an image field using this function, be sure that PSIMAGEVER field is also on the same record as the image field being updated.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll area in the component buffer.
target_row	The row number of the target row.
[*recordname* **.**]*fieldname*	The name of the field associated with the image file. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the function call is in a record definition other than *recordname*.

Returns

Returns a Boolean value: True if image was successfully deleted, False otherwise.

Example

```
&Rslt = DeleteImage(EMPL_PHOTO.EMPLOYEE_PHOTO);
```

See Also

Chapter 1, PeopleCode Built-in Functions, InsertImage , page 295.

DeleteRecord

Syntax

DeleteRecord(*level_zero_recfield*)

Description

Use **DeleteRecord** to remove a high-level row of data and all dependent rows in other tables from the database.

Note. This function remains for backward compatibility only. Use the **Delete** Record class method instead.

DeleteRecord deletes the component's level-zero row from the database**,** deletes any dependent rows in other tables from the database, and exits the component.

This function, like **DeleteRow**, initially marks the record or row as needing to be deleted. At save time the row is actually deleted from the database and cleared from the buffer.

This function works only if the PeopleCode is on a level-zero field. It cannot be used from SavePostChange or WorkFlow PeopleCode.

See Also

Chapter 25, Record Class , Delete, page 1447, *PeopleCode Developer's Guide*, Data Buffer Access.

Parameters

Parameter	Description
level_zero_recfield	A *recordname*.*fieldname* reference identifying any field on the level-zero area of the page.

Returns

Optionally returns a Boolean value indicating whether the deletion was completed successfully.

Example

The following example, which is in SavePreChange PeopleCode on a level-zero field, deletes the high-level row and all dependent rows in other tables if the current page is EMPLOYEE_ID_DELETE.

```
if %Page = PAGE.EMPLOYEE_ID_DELETE then
     &success = DeleteRecord(EMPLID);
end-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, DeleteRow , page 140.

DeleteRow

Syntax

DeleteRow(*scrollpath*, *target_row*)

Where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD**.*target_recname*

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

Use **DeleteRow** to delete rows programmatically.

Note. This function remains for backward compatibility only. Use the **DeleteRow** Rowset class method instead.

A call to this function causes the RowDelete event sequence to fire, as if an user had manually deleted a row.

DeleteRow cannot be executed from the same scroll level where the deletion will take place, or from a lower scroll level. Place the PeopleCode in a higher scroll level record.

When DeleteRow is used in a loop, you have to process rows from high to low to achieve the correct results, that is, you must delete from the *bottom up* rather than from the top down. This is necessary because the rows are renumbered after they are deleted (if you delete row one, row two becomes row one).

See Also

Chapter 27, Rowset Class , DeleteRow, page 1484, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number of the row to delete.

Returns

Boolean (optional). DeleteRow returns a Boolean value indicating whether the deletion was completed successfully.

Example

In the following example DeleteRow is used in a **For** loop. The example checks values in each row, then conditionally deletes the row. Note the syntax of the For loop, including the use of the -1 in the **Step** clause to loop from the highest to lowest values. This ensures that the renumbering of the rows will not affect the loop.

```
For &L1 = &X1 To 1 Step - 1
    &SECTION = FetchValue(AE_STMT_TBL.AE_SECTION, &L1);
    &STEP = FetchValue(AE_STMT_TBL.AE_STEP, &L1);
    If None(&SECTION, &STEP) Then
       DeleteRow(RECORD.AE_STMT_TBL, &L1);
    End-If;
 End-For;
```

See Also

Chapter 1, PeopleCode Built-in Functions, InsertRow , page 297, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

DeleteSQL

Syntax

DeleteSQL([**SQL**.]*sqlname*[, *dbtype*[, *effdt*]])

Description

The **DeleteSQL** function enables you to programmatically delete a SQL definition. The SQL definition must have already been created and saved, either using the CreateSQL and StoreSQL functions, or by using Application Designer.

When you create a SQL definition, you must create a base statement before you can create other types of statements, that is, one that has a *dbtype* as GENERIC and *effdt* as the null date (or Date(19000101)). If you specify a base (generic) statement to be deleted, *all* statements as well as the generic statement are deleted.

If you specify a non-generic statement that ends up matching the generic statement, DeleteSQL does *not* delete anything, and returns False.

You must commit all database changes prior to using this function. This is to avoid locking critical Tools tables and hence freezing all other users. You receive a runtime error message if you try to use this function when there are pending database updates, and your PeopleCode program terminates. You need to commit any database updates prior to using this function. The CommitWork PeopleCode function has been enhanced to allow this.

Parameters

Parameter	Description
sqlname	Specify the name of a SQL definition. This is either in the form SQL.*sqlname* or a string value giving the *sqlname*.
dbtype	Specify the database type associated with the SQL definition. This parameter takes a string value. If *dbtype* isn't specified or is null (""), it set by default to the current database type (the value returned from the %DbName system variable.) Valid values for *dbtype* are as follows. These values are not case sensitive: • APPSERVER • DB2 • DB2UNIX • INFORMIX • MICROSOFT • ORACLE • SYBASE **Note.** Database platforms are subject to change.
effdt	Specify the effective date associated with the SQL definition. If *effdt* isn't specified, it is set by default to the current as of date, that is, the value returned from the %AsOfDate system variable.

Returns

A Boolean value: True if the delete was successful, False if the specified SQL statement wasn't found, and terminates with an error message if there was another problem (that is, date in incorrect format, and so on.)

Example

The following code deletes the ABCD_XY SQL definition for the current DBType and as of date:

```
&RSLT = DeleteSQL(SQL.ABC_XY);

If NOT(&RSLT) Then
   /* SQL not found – do error processing */
End-if;
```

The following code deletes the ABCD_XY SQL Definition for the current DBType and November 3, 1998:

```
&RSLT = DeleteSQL(SQL.ABCD_XY, "",Date(19981103));
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109, Chapter 1, PeopleCode Built-in Functions, FetchSQL , page 180, Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271, Chapter 1, PeopleCode Built-in Functions, StoreSQL , page 450, Chapter 1, PeopleCode Built-in Functions, CommitWork , page 71, Chapter 31, SQL Class , page 1613, Chapter 3, System Variables , %DbName , page 567, Chapter 3, System Variables , %AsOfDate , page 563.

DeleteSystemPauseTimes

Syntax

DeleteSystemPauseTimes(*StartDay*, *StartTime*, *EndDay*, *EndTime*)

Description

The **DeleteSystemPauseTimes** function enables you to delete pause times that occur on your system by adding a row to the system pause times table.

This function is used in the PeopleCode for the Message Monitor. Pause times are set up in the Message Monitor.

Parameters

Parameter	Description
StartDay	Specify a number from 0-6. The values are:

Value	Description
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

Parameter	Description
StartTime	Specify a time, in seconds, since midnight.
EndDay	Specify a number from 0-6. The values are:

Value	Description
0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday

Parameter	Description
EndTime	Specify a time, in seconds, since midnight.

Returns

A Boolean value: True if the system pause time specified was deleted, False otherwise.

Example

```
Component Boolean &spt_changed;

/* deleting a system pause time interval; */

If Not DeleteSystemPauseTimes(PSSPTIMES.STARTINGDAY, PSSPTIMES.STARTINGSECOND,⇒
 PSSPTIMES.ENDINGDAY, PSSPTIMES.ENDINGSECOND) Then
   Error MsgGetText(117, 15, "");
Else
   &spt_changed = True;

/* to force a save; */

   PSSPTIMES.MSGSPTNAME = " ";

   DoSave();
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, AddSystemPauseTimes , page 37, *PeopleSoft Integration Broker*, Using Integration Broker Monitor.

DeQueue

Syntax

DeQueue(*physical queue ID*, *task type*, *task number*, *agent ID*)

Description

Once a task that has been placed in a queue by **EnQueue** and has been completed by the agent, use**DeQueue** to notify the queue server. The queue server removes the task from the queue and subtracts the cost of that task from the agent's workload.

Note. The queue server does not allow a task to be dequeued if the agent who owns that task is not logged on to that particular queue server. PeopleSoft recommends that you only use **DeQueue** in application pages that the MultiChannel Framework Console launches when agents accept or activate assigned tasks.

Parameters

Parameter	Description
physical queue ID	The physical queue is the internal representation of the logical queue that the agent signs onto and to which the task currently belongs. This is a string value, such as "sales3" or "marketing2." You retrieve the current physical queue from the query string in the URL of the page launched by the MultiChannel Framework console, using the Request class **GetParameter** method with the value `ps_qid`.
task type	Specifies the type of task that an agent completed. It is a string value. The valid values are: • email • generic **Note.** This parameter is valid only for persistent tasks (email and generic). It is not valid for chat or voice tasks. You can retrieve the value from the query string in the URL of the application page launched by the MultiChannel Framework console. Use the Request class method **GetParameter** with the value `ps_tasktype`.
task number	Uniquely identifies a particular task. This is the task number returned by the **Enqueue** function when the system first inserted the task into a queue. This is a string value. You can retrieve the value from the query string in the URL of the application page launched by the MultiChannel Framework console. Use the Request class method **GetParameter**with the value `ps_tasknum`.
agent ID	Identifies the agent who processed the task. This is a string value. You can retrieve the value from the query string in the URL of the application page launched by the MultiChannel Framework console. Use the Request class method **GetParameter**with the value `ps_agentid`.

Returns

Returns 0 for success. Otherwise, it returns a message number. The message set ID for MultiChannel Framework is 162.

For example, 1302 is returned when an invalid task type or no value is provided.

Example

```
PSMCFFUNCLIB.MCF_QUEUE.Value = %Request.GetParameter("ps_qid");
PSMCFFUNCLIB.MCF_TASKTYPE.Value = %Request.GetParameter("ps_tasktype");
PSMCFFUNCLIB.MCF_TASKNUM.Value = %Request.GetParameter("ps_tasknum");
PSMCFFUNCLIB.MCF_AGENTID.Value = %Request.GetParameter("ps_agentid");

&nret = DeQueue(PSMCFFUNCLIB.MCF_QUEUE, PSMCFFUNCLIB.MCF_TASKTYPE,⇒
 PSMCFFUNCLIB.MCF_TASKNUM, PSMCFFUNCLIB.MCF_AGENTID);

If &nret = 0 Then
      MessageBox(0, "", 0, 0, "Successfully dequeued.");
      End-If
```

DisableMenuItem

Syntax

DisableMenuItem(**BARNAME.***menubar_name*, **ITEMNAME.***menuitem_name*)

Description

DisableMenuItem disables (makes unavailable) the specified menu item. To apply this function to a pop-up menu, use the PrePopup Event of the field with which the pop-up menu is associated.

If you're using this function with a pop-up menu associated with a page (not a field), the earliest event you can use is the PrePopup event for the first "real" field on the page (that is, the first field listed in the Order view of the page in PeopleSoft Application Designer.)

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Parameters

Parameter	Description
menubar_name	Name of the menu bar that owns the menu item, or, in the case of pop-up menus, the name of the pop-up menu that owns the menu item.
menuitem_name	Name of the menu item.

Returns

None.

Example

```
DisableMenuItem(BARNAME.MYPOPUP1, ITEMNAME.DO_JOB_TRANSFER);
```

See Also

Chapter 1, PeopleCode Built-in Functions, EnableMenuItem , page 157, Chapter 1, PeopleCode Built-in Functions, HideMenuItem , page 287, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, PrePopup Event.

DiscardRow

Syntax

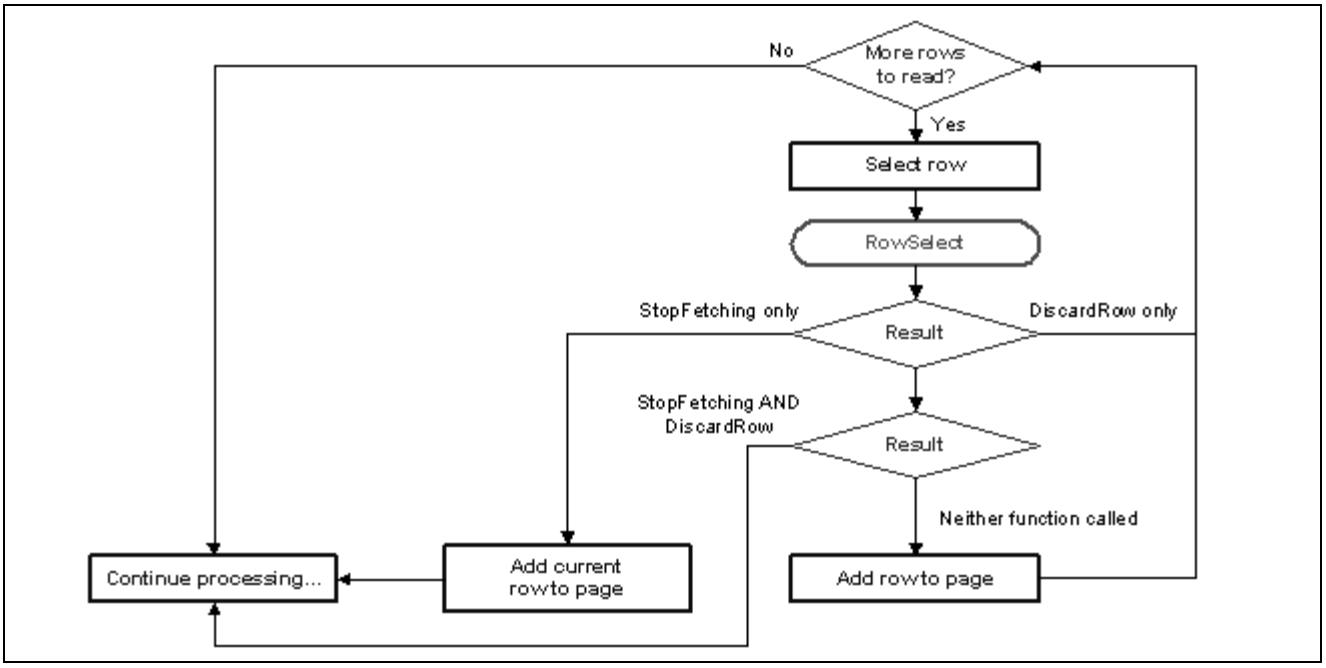
DiscardRow()

Description

DiscardRow prevents a row from being added to a page scroll during Row Select processing. This function is valid only in RowSelect PeopleCode. When DiscardRow is called during RowSelect processing, the current row is skipped (not added to the scroll). Processing then continues on the next row, unless **StopFetching** has also been called, in which case no more rows are added to the page.

DiscardRow has the same functionality as **Warning** in the RowSelect event. The anomalous behavior of Warning is supported for compatibility with previous releases of PeopleTools.

Note. Row Select processing is used infrequently, because it is more efficient to filter out rows of data using a search view or an effective-dated record before the rows are selected down to the client from the database server.



Flow of Row Select processing

Returns

None.

See Also

Chapter 1, PeopleCode Built-in Functions, StopFetching , page 449, Chapter 1, PeopleCode Built-in Functions, Warning , page 499, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Row Select Processing.

DoCancel

Syntax

```
DoCancel( )
```

Description

DoCancel cancels the current page as though the user had pressed ESC. On the Windows client, it terminates the current component and returns the user to the current menu with no component active (that is, to reset state). In Web client applications, DoCancel does one of the following:

- In the Page Applet DoCancel terminates the current component and returns the user to the search dialog box.
- In the Menu Applet DoCancel terminates the current component and returns the user to the current menu with no component active. This is the same behavior as on the Windows client.

DoCancel terminates any PeopleCode programs executing prior to a save action. It does not stop processing of PeopleCode in SaveEdit, SavePreChange, and SavePostChange events.

Returns

None.

See Also

Chapter 1, PeopleCode Built-in Functions, DoSave , page 154, Chapter 1, PeopleCode Built-in Functions, DoSaveNow , page 155, Chapter 1, PeopleCode Built-in Functions, DoModal , page 148, Chapter 1, PeopleCode Built-in Functions, EndModal , page 161, Chapter 1, PeopleCode Built-in Functions, WinEscape , page 502.

DoModal

Syntax

```
DoModal(PAGE.pagename, title, xpos, ypos,
        [level, scrollpath, target_row])
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_
row, ]] RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

The **DoModal** function displays a secondary page. Secondary pages are modal, meaning that the user must dismiss the secondary page before continuing work in the page from which the secondary page was called.

If you call DoModal without specifying a level number or any record parameters, the function uses the current context as the parent.

You can alternately specify a secondary page in a command push button definition without using PeopleCode. This may be preferable for performance reasons, especially with Web client code.

DoModal can display on a single page. To display an entire component modally, use DoModalComponent.

Any variable declared as a Component variable will still be defined after using a DoModal function.

See *PeopleSoft Application Designer*, Using Page Controls, Using Secondary Pages.

Restrictions on Use in PeopleCode Events

Control does not return to the line after DoModal until after the user has dismissed the secondary page. This interruption of processing makes DoModal a "think-time" function, which means that it shouldn't be used in any of the following PeopleCode events:

- SavePreChange.
- SavePostChange.
- Workflow.
- RowSelect.

- Any PeopleCode event that fires as a result of a ScrollSelect (or one of its relatives) function calls, or a Select (or one of its relatives) Rowset class method.

In addition, you can't use this function in the SearchInit event.

See *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Think-Time Functions.

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Considerations Using Exceptions and the DoModal Function

Using the **DoModal** function inside a try block *not* catch PeopleCode exceptions thrown in the new component. Starting a new component starts a brand new PeopleCode evaluation context. Catches are only caught for exceptions thrown within the *current* component.

In the following code example, the catch statement only catches exceptions thrown in the code *prior to* the DoModal, but not any exceptions that are thrown within the new component.

```
/* Set up transaction */
If %CompIntfcName = "" Then
   try
      &oTrans = &g_ERMS_TransactionCollection.GetTransactionByName(RB_EM_⇒
WRK.DESCR);
      &sSearchPage = &oTrans.SearchPage;
      &sSearchRecord = &oTrans.SearchRecord;
      &sSearchTitle = &oTrans.GetSearchPageTitle();
      If Not All(&sSearchPage, &sSearchRecord, &sSearchTitle) Then
         Error (MsgGetText(17834, 8081, "Message Not Found"));
      End-If;
      &c_ERMS_SearchTransaction = &oTrans;

      /* Attempt to transfer to hidden search page with configurable filter */
      &nModalReturn = DoModal(@("Page." | &sSearchPage), &sSearchTitle, - 1, - 1);
   catch Exception &e
      Error (MsgGetText(17834, 8082, "Message Not Found"));
   end-try;
```

Parameters

Parameter	Description
pagename	The name of the secondary page.
title	The text that displays in the caption of the secondary page.
xpos	The pixel coordinates of the top left corner of the secondary page, offset from the top left corner of the parent page (the default of -1, -1 means centered).
ypos	The pixel coordinates of the top right corner of the secondary page, offset from the top right corner of the parent page (the default of -1, -1 means centered).
level	Specifies the level of the scroll level on the parent page that contains the row corresponding to level 0 on the secondary page.
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number of the row in the parent page corresponding to the level 0 row in the secondary page.

Returns

Returns a number that indicates how the secondary page was terminated. A secondary page can be terminated by the user clicking a built-in OK or Cancel button, or by a call to the **EndModal** function in a PeopleCode program. In either case, the return value of DoModal is one of the following:

- 1 if the user clicked OK in the secondary page, or if 1 was passed in the EndModal function call that terminated the secondary page.
- 0 if the user clicked Cancel in the secondary page, or if 0 was passed in the EndModal function call that terminated the secondary page.

Example

```
DoModal(PAGE.EDUCATION_DTL, MsgGetText(1000, 167, "Education Details - %1",⇒
 EDUCATN.DEGREE), - 1, - 1, 1, RECORD.EDUCATN, CurrentRowNumber());
```

See Also

Chapter 1, PeopleCode Built-in Functions, EndModal , page 161, Chapter 1, PeopleCode Built-in Functions, IsModal , page 307, Chapter 1, PeopleCode Built-in Functions, DoModalComponent , page 150, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

DoModalComponent

Syntax

DoModalComponent(**MENUNAME.***menuname*, **BARNAME.***barname*, **ITEMNAME.***menuitem_name*, **PAGE.***component_item_name*, *action*, **RECORD.***shared_record_name* [, *keylist*])

where *keylist* is a list of field references in the form:
[*recordname.*]*field1* [, [*recordname.*]*field2*]. . .

Or

&RecordObject1 [, *&RecordObject2*]. . .

Description

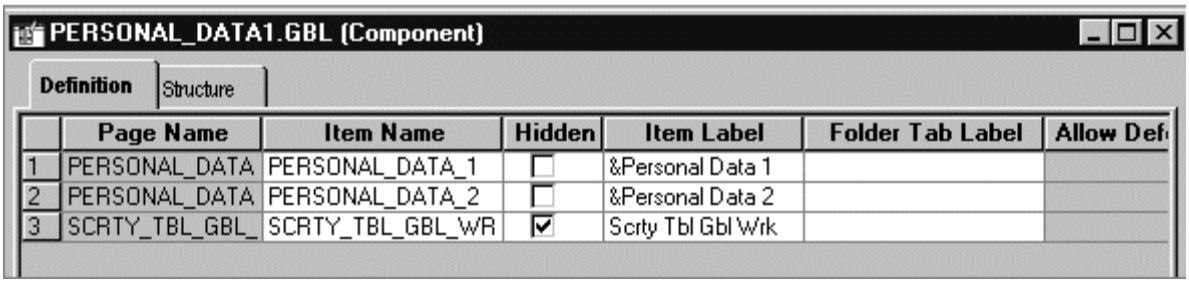
DoModalComponent launches a modal component. The modal component launches from within an originating component. After the modal component displays, the user can't proceed with changes to the originating component until either accepting or canceling the modal component.

Modal components can be displayed in any of the following action modes: Add, Update/Display, Update/Display All, Correction. A modal component can be launched from any component, including another modal component. You can use DoModalComponent from a secondary page.

The originating component and the modal component share data, including search keys, using a Shared Work Record or the values in the *fieldlist* parameter. If valid search keys are provided in the shared work record and populated with valid values before launching the modal component, the search is conducted using the provided search key values. If the *fieldlist* parameter isn't used and no search keys are provided, or if search key fields contain invalid values, the user accesses the modal component using a search dialog box.

Note. The user may see a different title for a search page if they enter the search page using this function versus from the regular navigation.

In the *component_item_name* parameter, make sure to pass the component item name for the page, not the page name. The component item name is specified in the Component Definition, in the Item Name column on the row corresponding to the specific page, as shown here:



Component Item Name parameter

Shared Work Records

The originating component and the modal component share fields in a Derived/Work record called a *shared work record*. Shared fields from this record must be placed at level zero of both the originating component and the modal component.

You can use the shared fields to:

- Pass values that are assigned to the search keys in the modal component search record. If these fields are missing or not valid, the search dialog box appears, enabling the user to enter search keys.
- Optionally pass other values from the originating component to the modal component.
- Pass values back from the modal component to the originating component for processing.

To do this, you have to write PeopleCode that:

- Assigns values to fields in the shared work record in the originating page at some point before the modal transfer takes place.

- Accesses and changes the values, if necessary, in the modal component.
- Accesses the values from the shared work record from the originating component after the modal component is dismissed.

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Restrictions on Use With SearchInit Event

You can't use this function in a SearchInit PeopleCode program.

Considerations Using Exceptions and the DoModalComponent Function

Using the **DoModalComponent** function inside a try block *not* catch PeopleCode exceptions thrown in the new component. Starting a new component starts a brand new PeopleCode evaluation context. Catches are only caught for exceptions thrown within the *current* component.

In the following code example, the catch statement only catches exceptions thrown in the code *prior to* using a DoModal function, but not any exceptions that are thrown within the new component.

```
/* Set up transaction */
If %CompIntfcName = "" Then
   try
      &oTrans = &g_ERMS_TransactionCollection.GetTransactionByName(RB_EM_⇒
WRK.DESCR);
      &sSearchPage = &oTrans.SearchPage;
      &sSearchRecord = &oTrans.SearchRecord;
      &sSearchTitle = &oTrans.GetSearchPageTitle();
      If Not All(&sSearchPage, &sSearchRecord, &sSearchTitle) Then
         Error (MsgGetText(17834, 8081, "Message Not Found"));
      End-If;
      &c_ERMS_SearchTransaction = &oTrans;

      /* Attempt to transfer to hidden search page with configurable filter */
      &nModalReturn = DoModal(@("Page." | &sSearchPage), &sSearchTitle, - 1, - 1);
   catch Exception &e
      Error (MsgGetText(17834, 8082, "Message Not Found"));
   end-try;
```

Parameters

Parameter	Description
menuname	Name of the menu through which the modal component is accessed.
barname	Name of the menu bar through which the modal component is accessed.
menuitem_name	Name of the menu item through which the modal component is accessed.
component_item_name	The component item name of the page to be displayed on top of the modal component when it displays. The component item name is specified in the component definition.
action	String representing the action mode in which to start up the component. You can use either a character value (passed in as a string) or a constant:

Numeric Value	Constant Value	Description
A	%Action_Add	Add
U	%Action_UpdateDisplay	Update/Display
L	%Action_UpdateDisplayAll	Update/Display All
C	%Action_Correction	Correction
E	%Action_DataEntry	Data Entry
P	%Action_Prompt	Prompt

Parameter	Description
	If only one action mode is allowed for the component, that action mode is used. If more than one action mode is allowed, the user can select which mode to come up in.
shared_record_name	The record name of the shared work record (preceded by the reserved word **Record**). This record must include: • Fields that are search keys in the modal component search record; if search key fields are not provided, or if they are invalid, the user accesses the modal component using the search dialog box. • Other fields to pass to the modal component. • Fields to get back from the modal component after it has finished processing.
keylist	An optional list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. If a record object is specified, any field of that record object *that is also a field of the search record for the destination component* is added to *keylist*. The keys in *fieldlist* must uniquely identify a row in the "to" page search record. If a unique row is not identified, the search dialog box appears. If the *keylist* parameter is not supplied then the destination components' search key must be found as part of the source component's level 0 record buffer.

Returns

Returns a Boolean that indicates how the secondary page was terminated. A secondary page can be terminated by the user clicking a built-in OK or Cancel button, or by a call to the **EndModal** function in a PeopleCode program. In either case, the return value of DoModalComponent is one of the following:

- True if the user clicked OK in the secondary page, or if 1 was passed in the EndModal function call that terminated the secondary page.
- False if the user clicked Cancel in the secondary page, or if 0 was passed in the EndModal function call that terminated the secondary page.

Example

The following example shows how to structure a DoModalComponent function call:

```
DoModalComponent(MENUNAME.MAINTAIN_ITEMS_FOR_INVENTORY, BARNAME.USE_A,⇒
 ITEMNAME.ITEM_DEFINITION, COMPONENT.ESTABLISH_AN_ITEM, "C", RECORD.NEW7_WRK);
```

Supporting PeopleCode is required if you must assign values to fields in the shared work record or access those values, either from the originating component, or from the modal component.

See Also

Chapter 1, PeopleCode Built-in Functions, IsModalComponent , page 308, Chapter 1, PeopleCode Built-in Functions, Transfer , page 463, Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475, Chapter 1, PeopleCode Built-in Functions, EndModalComponent , page 162, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Implementing Modal Transfers.

DoModalPanelGroup

Syntax

DoModalPanelGroup(**MENUNAME.***menuname*, **BARNAME.***barname*, **ITEMNAME.***menuitem_name*, **PANEL.***panel_group_item_name*, *action*, **RECORD.***shared_record_name*)

Description

DoModalPanelGroup launches a modal component.

Note. The **DoModalPanelGroup** function is supported for compatibility with previous releases of PeopleTools. Future applications should use DoModalComponent instead.

See Also

Chapter 1, PeopleCode Built-in Functions, DoModalComponent , page 150.

DoSave

Syntax

DoSave()

Description

DoSave saves the current page. DoSave defers processing to the end of the current PeopleCode program event, as distinct from DoSaveNow, which causes save processing (including SaveEdit, SavePreChange, SavePostChange, and Workflow PeopleCode) to be executed immediately.

DoSave can be used only in FieldEdit, FieldChange, or MenuItemSelected PeopleCode.

Parameters

None.

Returns

None.

Example

The following example sets up a key list with AddKeyListItem, saves the current page, and then transfers the user to a page named PAGE_2.

```
ClearKeyListItem( );
AddKeyListItem(OPRID, OPRID);
AddKeyListItem(REQUEST_ID, REQUEST_ID);
SetNextPage("PAGE_2");
DoSave( );
TransferPage( );
```

See Also

Chapter 1, PeopleCode Built-in Functions, DoCancel , page 147, Chapter 1, PeopleCode Built-in Functions, DoSaveNow , page 155, Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475, Chapter 1, PeopleCode Built-in Functions, AddKeyListItem , page 36.

DoSaveNow

Syntax

DoSaveNow()

Description

DoSaveNow is designed primarily for use with remote calls. It enables a PeopleCode program to save page data to the database before running a remote process (most frequently a COBOL process) that will access the database directly. It is generally necessary to call DoSaveNow before calling **RemoteCall**.

DoSaveNow causes the current page to be saved immediately. Save processing (including SaveEdit, SavePreChange, SavePostChange, and Workflow PeopleCode) is executed before continuing execution of the program where DoSaveNow is called. DoSaveNow differs from the **DoSave** function in that DoSave defers saving the component until after any running PeopleCode is completed.

DoSaveNow can only be called from a FieldEdit or FieldChange event.

See Chapter 1, PeopleCode Built-in Functions, RemoteCall , page 364 and Chapter 1, PeopleCode Built-in Functions, DoSave , page 154.

Errors in DoSaveNow Save Processing

DoSaveNow initiates save processing. It handles errors that occur during save processing as follows:

- If save processing encounters a SaveEdit error, it displays the appropriate message box. DoSaveNow immediately exits from the originating FieldChange or FieldEdit program. The user can correct the error and continue.

- If save processing encounters a fatal error, it displays the appropriate fatal error. DoSaveNow handles the error by immediately exiting from the originating FieldChange or FieldEdit program. The user must then cancel the page.
- If save processing completes with no errors, PeopleCode execution continues on the line after the DoSaveNow call in FieldChange or FieldEdit.

Restrictions on Use of DoSaveNow

The following restrictions apply:

- DoSaveNow can be executed only from a FieldEdit or FieldChange event.
- DoSaveNow is only allowed prior to the first CallAppEngine function in a FieldChange event, but not afterward.
- Deferred operations should not be called before the DoSaveNow. Deferred operations include **DoSave, TransferPage**, **SetCursorPos**, **EndModal**.
- Components that use DoSaveNow must not have **DoCancel**, **Transfer**, **TransferPage**, or **WinEscape** calls in PeopleCode attached to save action events (SaveEdit, SavePreChange, and SavePostChange), because these functions terminate the component, which would cause the program from which **DoSaveNow** was called to terminate prematurely.

Note. Application developers should be aware that DoSaveNow may result in unpredictable behavior if PeopleCode in save events deletes rows or inserts rows into scrolls. PeopleCode that runs after DoSaveNow must be designed around the possibility that rows were deleted or inserted (which causes row number assignments to change). Modifying data on a deleted row may cause it to become “undeleted.”

Returns

None.

Example

The following example calls DoSaveNow to save the component prior to running a remote process in the **remote_depletion** declared function:

```
Declare Function remote_depletion PeopleCode FUNCLIB_ININTFC.RUN_DEPLETION ⇒
FieldFormula;

/*
Express Issue Page - run Depletion job through RemoteCall()
*/
If %Component = COMPONENT.EXPRESS_ISSUE_INV Then
   DoSaveNow();
   &BUSINESS_UNIT = FetchValue(SHIP_HDR_INV.BUSINESS_UNIT, 1);
   &SHIP_OPTION = "S";
   &SHIP_ID = FetchValue(SHIP_HDR_INV.SHIP_ID, 1);
   remote_depletion(&BUSINESS_UNIT, &SHIP_OPTION, &SHIP_ID, &PROGRAM_STATUS);
End-If
```

EnableMenuItem

Syntax

EnableMenuItem(**BARNAME.***menubar_name*, **ITEMNAME.***menuitem_name*)

Description

EnableMenuItem enables (makes available for selection) the specified menu item. To apply this function to a pop-up menu, use the PrePopup Event of the field with which the pop-up menu is associated.

If you're using this function with a pop-up menu associated with a page (not a field), the earliest event you can use is the PrePopup event for the first "real" field on the page (that is, the first field listed in the **Order** view of the page in Application Designer.)

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Parameters

Parameter	Description
menubar_name	Name of the menu bar that owns the menu item, or, in the case of pop-up menus, the name of the pop-up menu that owns the menu item.
menuitem_name	Name of the menu item.

Returns

None.

Example

```
EnableMenuItem(BARNAME.MYPOPUP1, ITEMNAME.DO_JOB_TRANSFER);
```

See Also

Chapter 1, PeopleCode Built-in Functions, DisableMenuItem , page 146, Chapter 1, PeopleCode Built-in Functions, HideMenuItem , page 287, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, PrePopup Event.

EncodeURL

Syntax

EncodeURL(*URLString*)

Description

The **EncodeURL** function applies URL encoding rules, including escape characters, to the string specified by *URLString*. The method used to encode the *URLString* is the standard defined by W3C. This function returns a string that contains the encoded URL. Special characters (/, :, and so on) are assumed to be used for special purposes (that is, as part of the URI/protocol information) and are not encoded. If you need to encode such characters, use the **EncodeURLForQueryString** function.

Parameters

Parameter	Description
URLString	Specify the string you want encoded. This parameter takes a string value.

Returns

An encoded string.

Example

```
&MYSTRING = EncodeURL("http://corp.office.com/hr/benefits/401k/401k_home.htm");
```

See Also

Chapter 1, PeopleCode Built-in Functions, EncodeURLForQueryString , page 158, Chapter 1, PeopleCode Built-in Functions, Unencode , page 483, Chapter 15, Internet Script Classes (iScript) , page 931.

EncodeURLForQueryString

Syntax

EncodeURLForQueryString(*URLString*)

Description

The **EncodeURLForQueryString** function encodes *URLString* for use in a query string parameter in an URL. It encodes &, ? and = characters, and all other unsafe characters.

If the link is constructed in a PeopleSoft Internet Architecture page, and the value is a link field, you should not call EncodeURL to encode the entire URL, as PIA does this for you. You still need to unencode the parameter value when you retrieve it, however.

Parameters

Parameter	Description
URLString	Specify the string you want encoded. This parameter takes a string value.

Returns

An encoded URL string.

Example

In an Internet Script, to construct an anchor with a URL in a query string parameter, do the following:

```
&url = "http://host/iclientservlet.jrun/peoplesoft8?emplid=1111&mkt=usa"

&href = %Request.RequestURI | "?" | %Request.QueryString | "&myurl=" | ⇒
EncodeURLForQueryString(&url);

%Response.WriteLine("<a href= " | EncodeURL(&href) | ">My Link</a>");
```

The following uses a generic method to find, then encode, the URL, for the external link:

```
&StartPos = Find("?", &URL, 1);
&CPColl = &Portal.ContentProviders;
&strHREF = EncodeURLForQueryString(Substring(&URL, &StartPos + 1, Len(&URL)⇒
  -&StartPos));
&LINK = &Portal.GetQualifiedURL("PortalServlet", "PortalOriginalURL=" |⇒
 &CPColl.ItembyName(&CP_NAME).URI | "?" | &strHREF);
```

See Also

Chapter 1, PeopleCode Built-in Functions, EncodeURL , page 157, Chapter 1, PeopleCode Built-in Functions, Unencode , page 483, Chapter 1, PeopleCode Built-in Functions, EscapeHTML , page 167, Chapter 1, PeopleCode Built-in Functions, EscapeJavascriptString , page 168, Chapter 1, PeopleCode Built-in Functions, EscapeWML , page 169, Chapter 15, Internet Script Classes (iScript) , page 931.

Encrypt

Syntax

Encrypt(*KeyString, ClearTextString*)

Description

The **Encrypt** function encrypts a string. This function is generally used with merchant passwords.

The value you use for *KeyString* must be the same for **Decrypt** and **Encrypt**.

Parameters

Parameter	Description
KeyString	Specify the key used for encrypting the string. You can specify a NULL value for this parameter, that is, two quotation marks with no blank space between them ("").
ClearTextString	Specify the string you want encrypted.

Returns

An encrypted string.

Example

The following encrypts a field if it contains a value. It also removes any blanks either preceding or trailing the value.

```
If All(PSCNFMRCHTOKEN.WRKTOKEN) Then
   CMPAUTHTOKEN = Encrypt("", RTrim(LTrim(PSCNFMRCHTOKEN.WRKTOKEN)));
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Decrypt , page 133, Chapter 1, PeopleCode Built-in Functions, Hash , page 284.
PeopleTools Security PeopleBook

EncryptNodePswd

Syntax

EncryptNodePswd(*Password*)

Description

Use the **EncryptNodePswd** function to encrypt an Integration Broker node password.

Note. This function is generally used with Integration Broker node password encryption. This function should not be used casually, as once you encrypt your node password, there is no decrypt PeopleCode method.

Parameters

Parameter	Description
Password	Specify the Integration Broker node password that you want encrypted, as a string.

Returns

An encrypted password as a string.

Example

In the following example, the password is stored in the database in an encrypted form instead of as plain text:

```
PSMSGNODEDEFN.IBPASSWORD = EncryptNodePswd(RTrim(LTrim(PSNODE_WRK.WRKPASSWORD)));
```

See Also

PeopleSoft Integration Broker, Administering Basic Integrations, Configuring Nodes.

EndMessage

Syntax

EndMessage(*message, messagebox_title*)

Description

Note. EndMessage is obsolete and is supported only for backward compatibility. **MessageBox**, which can now be used to display informational messages in any PeopleCode event, should be used instead.

The **EndMessage** function displays a message at the end of a transaction, at the time of the database COMMIT. This function can be used only in SavePostChange PeopleCode.

When an **EndMessage** function executes, PeopleTools:

- Verifies that the function is in SavePostChange; if it is not, an error occurs and the function terminates.
- Displays the message.
- Terminates the SavePostChange PeopleCode program.

Because it terminates the SavePostChange program, EndMessage is always be the last statement executed in the program on the specific field and row where the EndMessage is called. For this reason, you must write the SavePostChange program so that all necessary processing takes place before the EndMessage statement. PeopleCode programs on other fields and rows execute as usual.

Parameters

Parameter	Description
message	A string that must be enclosed in quotes containing the message text you want displayed.
messagebox_title	A string that must be enclosed in quotes containing the title of the message. It appears in the message box title bar.

Returns

None.

Example

The following example is from SavePostChange event PeopleCode. It checks to see whether a condition is true, and if so, it displays a message and terminates the SavePostChange program. If the condition is false, then processing continues in the **Else** clause:

```
If BUSINESS_UNIT = BUS_UNIT_WRK.DEFAULT_SETID Then
         EndMessage(MsgGet(20000, 12, "Message not found in Message Catalog"),⇒
" ");
      Else
/* any other SavePostChange processing in Else clause */
```

See Also

Chapter 1, PeopleCode Built-in Functions, MessageBox , page 319, Chapter 1, PeopleCode Built-in Functions, WinMessage , page 505.

EndModal

Syntax

EndModal(*returnvalue*)

Description

The EndModal function closes a currently open secondary page. It is required only for secondary pages that do not have OK and Cancel buttons. If the secondary page has OK and Cancel buttons, then the function for exiting the page is built in and no PeopleCode is required.

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Parameters

Parameter	Description
returnvalue	A Number value that determines whether the secondary page data is copied back to the parent page. A positive value runs **SaveEdit** PeopleCode and copies the data (this is the same as pressing the **OK** button). A value of zero skips **SaveEdit** and discards buffer changes made in the secondary page (this is the same as pressing the **Cancel** button). This value becomes the return value of the **DoModal** function that started the secondary page, and it can be tested after the secondary page is closed.

Returns

None.

Example

The following statement acts as an OK button:

```
EndModal(1);
```

The following statement acts as a Cancel button:

```
EndModal(0);
```

See Also

Chapter 1, PeopleCode Built-in Functions, DoModal , page 148, Chapter 1, PeopleCode Built-in Functions, IsModal , page 307.

EndModalComponent

Syntax

EndModalComponent(*ReturnValue*)

Description

The **EndModalComponent** function closes a currently open secondary component. You could use this for creating your own links to exit a modal component.

Restrictions on Use With a Component Interface

This function can't be used by a PeopleCode program that's been called by a Component Interface, and is ignored.

Parameters

Parameter	Description
ReturnValue	A Number value that determines whether the modal component data is saved and copied back to the parent page. A positive value saves the data in the component to the database, including all save processing and PeopleCode (this is the same as pressing the OK button). It also copies the data in the shared work record, if any, back to the primary component. A value of zero skips save processing discards buffer changes made in the modal component (this is the same as pressing the Cancel button).

Returns

A Boolean value: True if a non-zero value was used, False if zero was used.

Example

```
EndModalComponent(0);  /* cancels the component without saving */

EndModalComponent(1); /* saves and closes the component */
```

See Also

Chapter 1, PeopleCode Built-in Functions, DoModalComponent , page 150, Chapter 1, PeopleCode Built-in Functions, EndModal , page 161, Chapter 1, PeopleCode Built-in Functions, IsModal , page 307.

EnQueue

Syntax

EnQueue(*logical queue*, *task type*, *Relative URL*, *Language_Code* [,*subject*][, *agent ID*][, *overflow timeout*][, *escalation timeout*][, *cost*][, *priority*][, *skill level*])

Description

EnQueue assigns a task to one of the active, physical queues belonging to the specified logical queue. The physical queue to which the system assigns the task is chosen randomly to balance load across the queues.

Note. PeopleSoft recommends that you always follow the **EnQueue** function with the **NotifyQ** function.

See Chapter 1, PeopleCode Built-in Functions, NotifyQ , page 336.

Parameters

Parameter	Description
logical queue ID	Specifies the logical queue in which the task should be queued. It is a string value. The logical queue ID is a case-sensitive value. The case used in the **EnQueue** function must exactly match the case used when creating the logical queue ID with the MultiChannel Framework administration pages.
task type	Specifies the type of task to be inserted. It is a string value. The valid values are: • email • generic **Note.** This parameter does not apply to voice or chat. Chat tasks are enqueued using **InitChat**. Voice queueing is managed by PeopleSoft CTI.
Relative URL	The system uses this relative URL to generate the URL of the appropriate application page for the MultiChannel Framework console to launch when an agent accepts this task. The application page should contain the logic to enable the agent to resolve the task and either forward the task using **Forward** or dequeue the task using **DeQueue**.

Parameter	Description
Language_Code	This is the language code associated with the task to be enqueued. It is a string value that must exist in the PeopleSoft language table. The queue server only assigns this task to an agent whose list of languages contains this value. For example if an email to be enqueued is written in English, the language code would be “ENG”, and this email would only be assigned to agents whose language list contains English.
subject	This is an optional parameter. It is a string value describing the purpose of the request. This value appears on the agent’s console when the system assigns the task.
agent ID	Specifies the assigned agent. This is an optional, string parameter. If specified, the system holds the task until the specified agent is available to take this task. If this parameter is left blank, the queue server assigns it to the first available agent. **Note.** For better performance, PeopleSoft recommends not specifying the target agent as this has a processing overhead for the queue servers and does not allow the system to balance workload across all available agents.
overflow timeout	This is an optional parameter. It is an integer value expressing the overflow timeout in minutes. The overflow timeout is the time period within which a queue server has to find an agent who accepts the task (clicks on the flashing icon on the MultiChannel console). If the task is not accepted within this time, the task is removed from the queue and placed in the MultiChannel overflow table. If you do not specify a value, the system uses the default value specified for that task type in the Task Configuration page.
escalation timeout	This is an optional parameter. It is an integer value expressing the escalation timeout in minutes. The escalation timeout is the time period within which a task must be completed by the agent and closed with **DeQueue**. If the task is not closed within this time, the task is removed from the queue and from the agent’s accepted task list, which means the task becomes unassigned. Then the task is placed in the MultiChannel Framework escalation table. If no value is specified, the system uses the default specified for that task type in the Task Configuration pages.
cost	This is an optional parameter. It is an integer value measuring the workload each task places on an agent. The cost of a task is an estimate of the tasks’s expected complexity and of the time required to resolve the task. The minimum value is 0, and there is no maximum value. The cost of a task is added to an agent’s workload after accepting a task on the MultiChannel Framework console. A task can’t be assigned to an agent if the difference between the current workload and the maximum workload defined for that agent on the Agent configuration page is less than the cost of this task. If you do not specify a value, the system uses the default value specified for that task in the Task Configuration pages. **Note.** If the required skill level or cost submitted exceeds the highest skill level or maximum workoad of any of the agents on that queue, the task cannot be assigned.

Parameter	Description
priority	This is an optional parameter. It is an integer value expressing the priority level of the request. The minimum value is 0 and there is no maximum value. A higher value means a higher priority. Tasks are ordered on a physical queue based on their assigned priority. That is, the system assigns a task of a higher priority before it assigns a task of a lower priority. If no value is specified, the system uses the default value specified for that task type in the Task Configuration page. When tasks have the same priority, the system orders the tasks according to the time they were created. For example, suppose the following tasks exist: Priority 2 created at 11:15 AM and Priority 2 created at 11:16 AM. In this case, the system assigns the task created at 11:15 AM before the task created at 11:16 AM.
skill level	This is an optional parameter. It is an integer value expressing the minimum skill level required of the agent to whom the system routes the request. You set an agent's skill level in the Agent configuration page. The queue server assigns this task type to an available agent on that queue whose lowest skill level is greater than or equal to the skill level required by the task. If no value is specified, the system uses the default value specified for that task type in the Task Configuration page. **Note.** If the required skill level or cost submitted exceeds the highest skill level or maximum workoad of any of the agents on that queue, the task cannot be assigned.

Returns

If the insert was successful, the function returns a task number in the form of a string.

If unsuccessful, it returns a message number. The message set ID for MultiChannel Framework is 162.

For example, 1302 is returned when an invalid task type or no value is provided.

Example

```
&PortalValue = Portal.EMPLOYEE;
   &NodeValue = Node.QE_LOCAL; /*If running in Application Engine, this code
assumes CONTENT URI has been set in node defn admin page*/

   &MyCompURL = GenerateComponentContentRelURL(&PortalValue, &NodeValue,
MenuName.PT_MCF, "GBL", Component.MCFEM_DEMOERMS_CMP, Page.MCFEM_ERMSMN, "");
   &MyCompURL = &MyCompURL | "&ps_emailid=" | &emailid; /*Query string
dependent on component. Our demo comonent just needs email id*/

rem The URL to be passed will look something like;
rem "/psc/ps/EMPLOYEE/QE_LOCAL/c/PT_MCF.MCF_DEMOERMS_CMP.GBL?Page=MCFEM_ERMSMN";

&strtasknum = EnQueue(&queueID, "email", &MyCompURL, &langcode,
&subject, "QEDMO", 15, 60, &cost, &priority, &minskill);
```

Error

Syntax

Error *str*

Description

Error is generally used in FieldEdit or SaveEdit PeopleCode to stop processing and display an error message. It is distinct from **Warning**, which displays a warning message, but does not stop processing. **Error** is also used in RowDelete and RowSelect PeopleCode events.

Warning! The behavior of the Error function in the RowSelect event is very different from its normal behavior.

See the Error in RowSelect section for more details.

The text of the error message (the *str* parameter), should always be stored in the Message Catalog and retrieved using the **MsgGet** or **MsgGetText** function. This makes it much easier for the text to be translated, and it also enables you to include more detailed Explain text about the error.

When Error executes in a PeopleCode program, the program terminates immediately and no statements after the Error are executed. In other respects behavior of Error differs, depending on which PeopleCode event the function occurs in.

Errors in FieldEdit and SaveEdit

The primary use of Error is in FieldEdit and SaveEdit PeopleCode:

- In FieldEdit, Error stops processing, displays a message, and highlights the relevant field.
- In SaveEdit, Error stops all save processing and displays a message, but does not highlight any field. You can move the cursor to a specific field using the **SetCursorPos** function, but be sure to call SetCursorPos *before* calling Error, otherwise Error stops processing before SetCursorPos is called. Note that an Error on any field in SaveEdit stops all save processing, and no page data is saved to the database.

Errors in RowDelete

When the user attempts to delete a row of data, the system first prompts for confirmation. If the user confirms, the RowDelete event fires. An Error in the RowDelete event displays a message and prevents the row from being deleted.

Error in RowSelect

The behavior of Error in RowSelect is totally anomalous, and is supported only for backward compatibility. It is used to filter rows that are being added to a page scroll after the rows have been selected and brought down to the client. *No message is displayed.* Error causes the Component Processor to add the current row (the one where the PeopleCode is executing) to the page scroll, then stops adding any additional rows to the page scroll.

The behavior of Error in the RowSelect event enables you to filter out rows that are above or below some limiting value. In practice this technique is rarely used, because it is more efficient to filter out rows of data before they are brought down to the client. This can be accomplished with search views or effective date processing.

Errors in Other Events

Do not use the Error function in any of the remaining events, which include:

- FieldDefault
- FieldFormula
- RowInit

- FieldChange
- Prepopup
- RowInsert
- SavePreChange
- SavePostChange

The user has no control over processing that occurs in these events. If the Component Processor encounters an Error in one of these events, the user can't fix it. The Component Processor requires the user to cancel the component to avoid unpredictable results, which results in loss of any changes that the user has made.

Parameters

Parameter	Description
Str	A string containing the text of the error message. This string should always be stored in the Message Catalog and retrieved using the MsgGet or MsgGetText function. This makes translation much easier and also enables you to provide detailed Explain text about the error.

Returns

None.

Example

The following example, from SaveEdit PeopleCode, displays an error message, stops all save processing, and places the cursor in the QTY_ADJUSTED field. Note that SetCursorPos must be called before Error.

```
If PAGES2_INV_WRK.PHYS_CYC_INV_FLG = "Y" Then
      SetCursorPos(%Page, PHYSICAL_INV.INV_LOT_ID, CurrentRowNumber(1), ⇒
QTY_ADJUSTED, CurrentRowNumber());
      Error MsgGet(11100, 180, "Message not found.");
   End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, MsgGet , page 328, Chapter 1, PeopleCode Built-in Functions, MsgGetText , page 330, Chapter 1, PeopleCode Built-in Functions, SetCursorPos , page 404, Chapter 1, PeopleCode Built-in Functions, Warning , page 499, Chapter 1, PeopleCode Built-in Functions, WinMessage , page 505.

EscapeHTML

Syntax

EscapeHTML(*HTMLString*)

Description

The **EscapeHTML** function takes *HTMLString* and replaces all characters that are significant to a browser with special HTML escape sequences.

The characters that are replaced are ones that cause the browser to misinterpret the HTML if they aren't encoded. This function is intended to make the text "browser safe".

For example, the less-than symbol (<) is replaced with `&lt.`, a single quotation mark (') is replaced with `&#039.`, and so on.

This function is for use with strings that display in an HTML area.

Parameters

Parameter	Description
HTMLString	Specify a string of HTML that contains characters that must be replaced with HTML escape sequences.

Returns

A string containing the original text plus HTML escape sequences.

See Also

Chapter 1, PeopleCode Built-in Functions, EscapeJavascriptString , page 168, Chapter 1, PeopleCode Built-in Functions, EscapeWML , page 169.

EscapeJavascriptString

Syntax

EscapeJavascriptString(*String*)

Description

The **EscapeJavascriptString** function takes *String* and replaces the characters that have special meaning in a JavaScript string with special escape sequences.

For example, a single quotation mark (') is replaced by \', a new line character is replaced by \n, and so on.

The characters that are replaced are ones that cause the browser to misinterpret the JavaScript if they aren't encoded. This function is intended to make the text "browser safe."

This function is for use with text that becomes part of a JavaScript program.

Parameters

Parameter	Description
String	Specify a string that contains character that need to be replaced with JavaScript escape sequences.

Returns

A string containing the original text plus JavaScript escape sequences.

See Also

Chapter 1, PeopleCode Built-in Functions, EscapeHTML , page 167, Chapter 1, PeopleCode Built-in Functions, EscapeWML , page 169.

EscapeWML

Syntax

EscapeWML(*String*)

Description

The EscapeWML function escapes special characters that are significant to WML. This includes <, >, $ (escaped as $$), &, ’ and ".

This function is for use with strings that display on an WML browser.

Parameters

Parameter	Description
String	Specify a string that contains characters that need to be replaced with WML escape sequences.

Returns

A string containing the original plus text plus WML escape sequences.

See Also

Chapter 1, PeopleCode Built-in Functions, EscapeHTML , page 167, Chapter 1, PeopleCode Built-in Functions, EscapeJavascriptString , page 168.

Evaluate

Syntax

```
Evaluate left_term
   When [relop_1] right_term_1
     [statement_list]
   .
   .
   .
  [When [relop_n] right_term_n
     [statement_list]]
   [When-other
      [statement_list]]
End-evaluate
```

Description

Use the **Evaluate** statement to check multiple conditions. It takes an expression, *left_term*, and compares it to compatible expressions (*right_term*) using the relational operators (*relop*) in a sequence of **When** clauses. If *relop* is omitted, then = is assumed. If the result of the comparison is True, it executes the statements in the When clause, then moves on to evaluate the comparison in the following When clause. It executes the statements in all of the When clauses for which the comparison evaluates to True. If and only if none of the When comparisons evaluates to True, it executes the statement in the **When-other** clause (if one is provided).

To end the Evaluate after the execution of a When clause, you can add a **Break** statement at the end of the clause.

Considerations Using When Clause

Generally, you use the When clause without a semicolon at the end of the statement. However, in certain circumstances, this can cause an error. For example, the following PeopleCode produces an error because the PeopleCode compiler cannot separate the end of the Where clause with the beginning of the next statement:

```
When = COMPONENT.GARBAGE

     (create BO_SEARCH:Runtime:BusinessContact_Contact(&fBusObjDescr, Null, ⇒
&fDerivedBOID, &fDerivedBORole, &fBusObjDescr1, Null, &fContactBOID, ⇒
&fContactRoleID, &fCustBOID, &fCustRoleID, "")).SearchItemSelected();

End-Evaluate;
```

If you place a semicolon after the When clause, the two expressions are read separately by the compiler. For example:

```
When = COMPONENT.GARBAGE;
```

Example

The following is an example of a When statement taken evaluates ACTION and performs various statements based on its value:

```
&PRIOR_STATUS = PriorEffdt(EMPL_STATUS);
Evaluate ACTION
When  "HIR"
   If %Mode = "A" Then
      Warning MsgGet(1000, 13, "You are hiring an employee and Action is not set⇒
 to Hire.");
   End-if;
   Break;
When = "REH"
   If All(&PRIOR_STATUS) and
         not (&PRIOR_STATUS = "T"  or
            &PRIOR_STATUS = "R" ) Then
      Error MsgGet(1000, 14, "Hire or Rehire action is valid
        only if employee status is Terminated or Retired.");
   End-if;
   Break;
When-Other
/* default code */
End-evaluate;
```

Exact

Syntax

Exact(*string1*, *string2*)

Description

Exact compares two text strings and returns True if they are the same, False otherwise. Exact is case-sensitive because it uses the internal character codes.

Returns

Returns a Boolean value: True if the two strings match in a case-sensitive comparison.

Example

The examples set &MATCH to True, then False:

```
&MATCH = Exact("PeopleSoft", "PeopleSoft");
&MATCH = Exact("PeopleSoft", "Peoplesoft");
```

See Also

Chapter 1, PeopleCode Built-in Functions, Len , page 313, Chapter 1, PeopleCode Built-in Functions, String , page 451, Chapter 2, Meta-SQL , %Substring , page 550.

Exec

Syntax

Exec(*command_str* [, *parameter*])

where *parameter* has one of the following formats:
Boolean constant
Exec_Constant + Path_Constant

Description

Exec is a cross-platform function that executes an external program on either UNIX or Windows.

This function has two parameter conventions in order to maintain upward compatibility with existing programs.

Note. All PeopleCode is executed on the application server. So if you're calling an interactive application, you receive an error. There shouldn't be any application interaction on the application server console.

The function can make either a synchronous or asynchronous call. Synchronous execution acts as a "modal" function, suspending the PeopleSoft application until the called executable completes. This is appropriate if you want to force the user (or the PeopleCode program) to wait for the function to complete its work before continuing processing. Asynchronous processing, which is the default, launches the executable and immediately returns control to the calling PeopleSoft application.

If Exec is unable to execute the external program, the PeopleCode program terminates with a fatal error. You may want to try to catch these exceptions by enclosing such statements in a try-catch statement (from the Exception Class).

Using an Absolute Path

If you don't specify anything for the second parameter, or if you specify a Boolean value, the path to PS_HOME is prefixed to the *command_str*.

If you specify constant values for the second parameter, PS_HOME may or may not be prefixed, depending on the values you select.

You can use the **GetEnv** function to determine the value of PS_HOME.

Restrictions on Use in PeopleCode Events

When Exec is used to execute a program synchronously (that is, if its *synch_exec* parameter is set to True) it behaves as a think-time function, which means that it can't be used in any of the following PeopleCode events:

- SavePreChange.
- SavePostChange.
- Workflow.
- RowSelect.
- Any PeopleCode event that fires as a result of a ScrollSelect (or one of its relatives) function calls, or a Select (or one of its relatives) Rowset class method.

See Also

PeopleCode Developer's Guide, Using Methods and Built-In Functions, Think-Time Functions.

Parameters

Parameter	Description
command_str	Specify a string containing the path and name of the executable file.
Boolean | *Constants*	If you specify a Boolean value, it indicates whether to execute the external program synchronously or asynchronously. Values are: • True - Synchronous • False - Asynchronous (default) If you don't specify a value, the program executes asynchronously. If you use this style, PS_HOME is always prefixed to *command_str.* If you specify constant values, you're specifying a numeric value composed of an *exec_constant* and a *path_constant*. The *exec_constant* specifies whether to execute the external program synchronously or not. The *path_constant* specifies how the path name is to be treated. The value specified is made up of the addition of these predefined constants. The function automatically converts the first token on *command_str* platform-specific separator characters to the appropriate form for where your PeopleCode program is executing, regardless of the *path_constant*. On a WIN32 system, a UNIX "/" separator is converted to "\", and on a UNIX system, a WIN32 "\" separator is converted to "/". This is only done for the first token on *command_str* assuming it to be some sort of file specification. This allows you to put file or program names in canonical form (such as, Unix style) as the first token on the exec command. For example: `&ExitCode = Exec("/ourbin/magicprogram ....", ⇒ %Exec_Synchronous + %FilePath_Absolute);` Values are:

Exec Constant	Description
%Exec_Asynchronous	Program executes asynchronously (the default)
%Exec_Synchronous	Program executes synchronously.

Path Constant	Description
%FilePath_Relative	PS_HOME is prefixed to *command_str*.
%FilePath_Absolute	Nothing is prefixed to *command_str*.

Returns

What is returned depends on what you specified for the second parameter.

If you specified a Boolean, a Number value equal to the process ID of the called process is returned.

If you specify constant values, the returned value contains the value of the exit code of the program executed using this function, unless you have executed the program asynchronously.

Example

```
&ExitCode = Exec("sh -c " | &scriptFile, %Exec_Synchronous + %FilePath_Absolute);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Declare Function , page 130, Chapter 1, PeopleCode Built-in Functions, RemoteCall , page 364, Chapter 1, PeopleCode Built-in Functions, WinExec , page 503, Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389, Chapter 1, PeopleCode Built-in Functions, ScrollSelectNew , page 392, Chapter 1, PeopleCode Built-in Functions, RowScrollSelect , page 384, Chapter 1, PeopleCode Built-in Functions, RowScrollSelectNew , page 386, Chapter 1, PeopleCode Built-in Functions, GetEnv , page 241, Chapter 27, Rowset Class , Select, page 1495, Chapter 11, Exception Class , page 835, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Think-Time Functions.

ExecuteRolePeopleCode

Syntax

ExecuteRolePeopleCode(*RoleName*)

Description

The **ExecuteRolePeopleCode** function executes the PeopleCode Rule for the Role *RoleName*. This function returns an array of string containing dynamic members (UserIds).

Typically, this function is used by an Application Engine process that runs periodically and executes the role rules for different roles. It could then write the results of the rules (a list of users) into the security tables, effectively placing users in certain roles based on the rule.

Parameters

Parameter	Description
RoleName	Specify the name of an existing role.

Returns

An array of string containing the appropriate UserIds.

Example

The following saves valid users to a temporary table:

```
Local array of string &pcode_array_users;

SQLExec("delete from ps_dynrole_tmp where ROLENAME=:1", &ROLENAME);
   If &pcode_rule_status = "Y" Then
      SQLExec("select RECNAME, FIELDNAME, PC_EVENT_TYPE, PC_FUNCTION_NAME from⇒
 PSROLEDEFN where ROLENAME= :1", &ROLENAME, &rec, &fld, &pce, &pcf);
      If (&rec <> "" And
            &fld <> "" And
            &pce <> "" And
            &pcf <> "") Then
         &pcode_array_users = ExecuteRolePeopleCode(&ROLENAME);
         &pcode_results = True;
      Else
         &pcode_results = False;
      End-If;
      &comb_array_users = &pcode_array_users;
   End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, ExecuteRoleQuery , page 174, Chapter 1, PeopleCode Built-in Functions, ExecuteRoleWorkflowQuery , page 175, Chapter 1, PeopleCode Built-in Functions, IsUserInPermissionList , page 312, Chapter 1, PeopleCode Built-in Functions, IsUserInRole , page 312, Chapter 3, System Variables , %Roles , page 577.
PeopleBook: PeopleSoft Security

ExecuteRoleQuery

Syntax

ExecuteRoleQuery(*RoleName*, *BindVars*)

where *BindVars* is an arbitrary-length list of bind variables that are stings in the form:
bindvar1 [, *bindvar2*]. . .

Description

The **ExecuteRoleQuery** function executes the Query rule for the role *rolename*, passing in *BindVars* as the bind variables. This function returns an array object containing the appropriate user members (UserIds).

Parameters

Parameter	Description
RoleName	Specify the name of an existing role.
BindVars	A list of bind variables to be substituted in the query. These bind variables must be strings. You can't use numbers, dates, and so on.

Returns

An array object containing the appropriate UserIds.

See Also

Chapter 1, PeopleCode Built-in Functions, ExecuteRolePeopleCode , page 173, Chapter 1, PeopleCode Built-in Functions, ExecuteRoleWorkflowQuery , page 175, Chapter 1, PeopleCode Built-in Functions, IsUserInPermissionList , page 312, Chapter 1, PeopleCode Built-in Functions, IsUserInRole , page 312, Chapter 3, System Variables , %Roles , page 577.
PeopleBook: PeopleSoft Security

ExecuteRoleWorkflowQuery

Syntax

```
ExecuteRoleWorkflowQuery(RoleName, BindVars)
```

where *BindVars* is an arbitrary-length list of bind variables in the form:

```
bindvar1 [, bindvar2]. . .
```

Description

The **ExecuteRoleWorkflowQuery** function executes the Workflow Query rule for the role *rolename*, passing in *BindVars* as the bind variables. This function returns an array object containing the appropriate user members (UserIds).

Parameters

Parameter	Description
RoleName	Specify the name of an existing role.
BindVars	A list of bind variables to be substituted in the query.

Returns

An array object containing the appropriate UserIds.

See Also

Chapter 1, PeopleCode Built-in Functions, ExecuteRolePeopleCode , page 173, Chapter 1, PeopleCode Built-in Functions, ExecuteRoleQuery , page 174, Chapter 1, PeopleCode Built-in Functions, IsUserInPermissionList , page 312, Chapter 1, PeopleCode Built-in Functions, IsUserInRole , page 312, Chapter 3, System Variables , %Roles , page 577.
PeopleBook: PeopleSoft Security

Exit

Syntax

Exit([1])

Description

Exit causes immediate termination of a PeopleCode program. If the Exit statement is executed within a PeopleCode function, the main program terminates.

Note. `Exit(1)` does *not* rollback iScript (XmlLink) transactions. To rollback in an XmlLink, you can use either the Error built-in function, or the MessageBox built-in function with a message error severity of error.

Parameters

Parameter	Description
1	Use this parameter to rollback database changes. Generally, this parameter is used in PeopleCode programs that affect messages. When used with a message, all database changes are rolled back, errors for the subscription contract are written to the subscription contract error table, and the status of the message is marked to **Error**. All errors that have occurred for this message are viewable in the message monitor: even those errors detected by **ExecuteEdits**.

Note. This function takes only numeric values. It fails if you use a Boolean value, True or False.

Returns

None.

Example

The following example terminates the main program from a **For** loop:

```
For &I = 1 To ActiveRowCount(RECORD.SP_BUIN_NONVW)
  &ITEM_SELECTED = FetchValue(ITEM_SELECTED, &I);
  If &ITEM_SELECTED = "Y" Then
      &FOUND = "Y";
      Exit;
  End-If;
End-For;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Break , page 49, Chapter 1, PeopleCode Built-in Functions, Return , page 376, *PeopleSoft Integration Broker*, Sending and Receiving Messages, Processing Inbound Errors.

Exp

Syntax

Exp(*n*)

Description

Exp returns **e** raised to the power of *n* where *n* is a number. The constant **e** equals 2.71828182845904, the base of natural logarithms. The number *n* is the exponent applied to the base **e**.

Exp is the inverse of **Ln**, which is the natural logarithm of x.

Returns

Returns a Number value equal to the constant e raised to the power of *n*.

Example

The examples set &NUM to 2.71828182845904, then 7.389056099(e2):

```
&NUM = Exp(1);
&NUM = Exp(2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Ln , page 315, Chapter 1, PeopleCode Built-in Functions, Log10 , page 316.

ExpandBindVar

Syntax

ExpandBindVar(*str*)

Description

Inline bind variables can be included in any PeopleCode string. An inline bind variable is a field reference (in the form *recordname.fieldname*), preceded by a colon. The inline bind variable references the value in the field.

ExpandBindVar expands any inline bind variables that it finds in *str* into strings (converting the data type of non-character fields) and returns the resulting string. This works with inline bind variables representing fields containing any data type except Object. It also expands bind variables specified using additional parameters.

See Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442.

Parameters

Parameter	Description
str	A string containing one or more inline bind variables.

Returns

Returns a String value equal to the input string with all bind variables expanded.

Example

A bind variable is included in the string &TESTSTR, which is then expanded into a new string containing the current value of BUS_EXPENSE_PER.EMPLID in place of the bind variable. If this program runs on the row for EMPLID 8001, the message displayed reads "This is a test using EmplID 8001".

```
&TESTSTR = "This is a test using EmplID :bus_expense_per.emplid";
```

```
&RESULT = ExpandBindVar(&TESTSTR);
WinMessage(&RESULT);
```

See Also

Chapter 1, PeopleCode Built-in Functions, MessageBox , page 319, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442.

ExpandEnvVar

Syntax

ExpandEnvVar(*string*)

Description

ExpandEnvVar converts any environment variables that it finds within *string* into String values and returns the entire resulting string.

Parameters

Parameter	Description
string	A string containing an environment variable.

Returns

Returns a string equal to *string* with any enclosed environment variables expanded to their values.

Example

Assume that the environment variable %NETDRIVE% is equal to "N:". The following PeopleCode sets *&newstring* equal to "The network drive is equal to N:":

```
&newstring = ExpandEnvVar("The network drive is equal to %netdrive%.");
```

See Also

Chapter 1, PeopleCode Built-in Functions, ExpandBindVar , page 177, Chapter 1, PeopleCode Built-in Functions, GetEnv , page 241, Chapter 1, PeopleCode Built-in Functions, GetCwd , page 241.

ExpandSqlBinds

Syntax

ExpandSqlBinds(*string*)

Description

Prior to PeopleTools 8.0, the PeopleCode replaced runtime parameter markers in SQL strings with the associated literal values. For databases that offer SQL statement caching, a match was never found in the cache so the SQL had to be re-parsed and re-assigned a query path.

To process skipped parameter markers, each parameter marker is assigned a unique number. This doesn’t change the values associated with the parameter markers.

However, *some SQL statements can’t contain parameter markers because of database compatibility.*

To process these exceptions, use the **ExpandSqlBinds** function. This function does the bind variable reference expansion, and can be used within a SQLExec statement or on its own.

You should use ExpandSQLBinds *only* for those parts of the string that you want to put literal values into. For example, you wouldn't use ExpandSQLBinds with any meta-SQL statements. The following code shows how to use ExpandSQLBinds with %Table:

```
SQLExec(ExpandSqlBinds("Insert....  Select A.Field, :1, :2 from ", "01", "02") |⇒
 "%table(:1)", Record.MASTER_ITEM_TBL);
```

Parameters

Parameter	Description
string	Specify the string you want to do the bind variable reference expansion on.

Returns

A string.

Example

The following example shows both the original string and what it expands to.

```
&NUM = 1;
&STRING = "My String";
&STR2 = ExpandSqlBinds("This :2 is an expanded string(:1)", &STRING, &NUM);
```

The previous code produces the following value for &STR2:

```
This 1 is an expanded string(My String)
```

If you're having problems with an old SQL statement binds, you can use ExpandSqlBinds with it. For example, if your SQLExec is this:

```
SQLExec("String with concatenated bindrefs 'M':2, 'M':1", &VAR1, &VAR2),⇒
 &FETCHRESULT1, &FETCHRESULT2);
```

you can make it work by expanding it as follows:

```
SQLExec(ExpandSqlBinds("String with concatenated bindrefs 'M':2, 'M':1", &VAR1,⇒
 &VAR2), &FETCHRESULT1, &FETCHRESULT2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442, Chapter 31, SQL Class , page 1613.

Fact

Syntax

Fact(*x*)

Description

Fact returns the factorial of a positive integer *x*. The factorial of a number *x* is equal to 1*2*3*...**x*. If *x* is not an integer, it is truncated to an integer.

Returns

Returns a Number equal to the factorial of x.

Example

The example sets &X to 1, 1, 2, then 24. **Fact**(2) is equal to 1*2; **Fact**(4) is equal to 1*2*3*4:

```
&X = Fact(0);
&X = Fact(1);
&X = Fact(2);
&X = Fact(4);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Product , page 349.

FetchSQL

Syntax

FetchSQL([**SQL.**]*sqlname*[, *dbtype*[, *effdt*]])

Description

The **FetchSQL** function returns the SQL definition with the given *sqlname* as SQL.*sqlname or* a string value, matching the *dbtype* and *effdt*. If *sqlname* is a literal name, it must be in quotes.

Parameters

Parameter	Description
sqlname	Specify the name of a SQL definition. This is either in the form SQL.*sqlname* or a string value giving the *sqlname*.
dbtype	Specify the database type associated with the SQL definition. This parameter takes a string value. If *dbtype* isn't specified or is null (""), it is set by default to the current database type (the value returned from the %DbType system variable.) Values for *dbtype* are as follows. These values are not case-sensitive: • APPSERVER • DB2 • DB2UNIX • INFORMIX • MICROSOFT • ORACLE • SYBASE **Note.** Database platforms are subject to change.
effdt	Specify the effective date associated with the SQL definition. If *effdt* isn't specified, it is set by default to the current as of date, that is, the value returned from the %AsOfDate system variable.

Returns

The SQL statement associated with *sqlname* as a string.

Example

The following code gets the text associated with the ABCD_XY SQL Definition for the current DBType and as of date:

```
&SQLSTR = FetchSQL(SQL.ABC_XY);
```

The following code gets the text associated with the ABCD_XY SQL Definition for the current DBType and November 3, 1998:

```
&SQLSTR = FetchSQL(SQL.ABCD_XY, "", Date(19981103));
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109, Chapter 1, PeopleCode Built-in Functions, DeleteSQL , page 141, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442, Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271, Chapter 1, PeopleCode Built-in Functions, StoreSQL , page 450, Chapter 31, SQL Class , page 1613, Chapter 3, System Variables , %AsOfDate , page 563, Chapter 3, System Variables , %DbName , page 567.

FetchValue

Syntax

FetchValue(*scrollpath*, *target_row*, [*recordname*.]*fieldname*)

where *scrollpath* is:

[**RECORD**.*level1_recname*, *level1_row*, [**RECORD**.*level2_recname*, *level2_row*,]] **RECORD**.*target_recname*

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

FetchValue returns the value of a buffer field in a specific row of a scroll level.

Note. This function remains for backward compatibility only. Use the **Value** Field class property instead.

This function is generally used to retrieve the values of buffer fields outside the current context; if a buffer field is in the current context, you can reference it directly using a [*recordname*.]*fieldname* expression.

See Also

Chapter 12, Field Class , Value, page 869, Chapter 12, Field Class , LongTranslateValue, page 862, Chapter 12, Field Class , ShortTranslateValue, page 866, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	An integer specifying the row on the target scroll level where the referenced buffer field is located.
[*recordname*.]*fieldname*	The name of the field where the value to fetch is located. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the call to FetchValue is in a record definition other than *recordname*.

Returns

Returns the field value as an Any data type.

Example

The following example retrieves the value from field DEPEND_ID in record DEPEND on row &ROW_CNT from scroll level one:

```
&VAL = FetchValue(SCROLL.DEPEND, &ROW_CNT, DEPEND.DEPEND_ID);
```

See Also

Chapter 1, PeopleCode Built-in Functions, ActiveRowCount , page 29, Chapter 1, PeopleCode Built-in Functions, CopyRow , page 90, Chapter 1, PeopleCode Built-in Functions, CurrentRowNumber , page 116, Chapter 1, PeopleCode Built-in Functions, PriorValue , page 349, Chapter 1, PeopleCode Built-in Functions, UpdateValue , page 490, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer.

FieldChanged

Syntax

The syntax of **FieldChanged** varies, depending on whether you want to use a scroll path reference or a contextual reference to specify the field.

If you want to use a scroll path reference, the syntax is:

FieldChanged(*scrollpath*, *target_row*, [*recordname*.]*fieldname*)

where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD**.*target_recname*

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.

If you want to use a contextual reference, the syntax is:

FieldChanged([*recordname*.]*fieldname*)

In this construction the scroll level and row number are determined based on the current context.

Description

FieldChanged returns True if the referenced buffer field has been modified since being retrieved from the database either by a user or by a PeopleCode program.

Note. This function remains for backward compatibility only. Use the **IsChanged** Field class property instead.

This is useful during SavePreChange or SavePostChange processing for checking whether to make related updates based on a change to a field.

See Also

Chapter 12, Field Class , IsChanged, page 857, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
[*recordname*.]*fieldname*	The name of the field where the value to check is located. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the call to **FieldChanged** is in a record definition other than *recordname*.
target_row	The row number of the target row. If this parameter is omitted, the function assumes the row on which the PeopleCode program is executing.

Related Functions

Save PeopleCode programs (SaveEdit, SavePreChange, SavePostChange) normally process each row of data in the component. The following four functions enable you to control more precisely when the Component Processor should perform save PeopleCode:

- FieldChanged
- RecordChanged
- RecordDeleted
- RecordNew

These functions enable you to restrict save program processing to specific rows.

Example

The following example checks three fields and sets a flag if any of them has changed:

```
/* Set the net change flag to 'Yes' if the scheduled date, scheduled */
/*  time or quantity requested is changed */
If FieldChanged(QTY_REQUESTED) Or
      FieldChanged(SCHED_DATE) Or
      FieldChanged(SCHED_TIME) Then
   NET_CHANGE_FLG = "Y";
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, RecordChanged , page 358, Chapter 1, PeopleCode Built-in Functions, RecordDeleted , page 359, Chapter 1, PeopleCode Built-in Functions, RecordNew , page 361.

FileExists

Syntax

```
FileExists(filename [, pathtype])
```

Description

The **FileExists** function determines whether a particular external file is present on your system, so you can decide which mode to use when you open the file for writing.

Note. If you want to open a file for reading, you should use the GetFile function or the Open File Class method with the "**e**" mode, which prevents another process from deleting or renaming the file between the time you tested for the file and when you open it.

Working With Relative Paths

If you specify a relative path, that path is appended to the path specified by an environmental variable, if the variable exists, in a particular order. This means if the first environmental variable in order is found, that is the one that's used. If it isn't found, the next one is searched for, and if found, is used, and so on.

The environmental variables that get used are different, depending on whether the machine where the code is running is an application server or not. If the code is running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]

If the code is *not* running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]
3. TEMP[\relative_path]

The PS_FILEDIR environmental variable is *not* initialized and set to a path automatically. If you want to use this environmental variable, you must set it yourself.

Note. Your system security should verify if a user has the correct permissions before allowing access to a drive (for example, if a user changes their **TEMP** environmental variable to a network drive they don't normally have access to, your system security should detect this.)

Parameters

Parameter	Description
filespec	Specify the name, and optionally, the path, of the file you want to test.
pathtype	If you have prepended a path to the file name, use this parameter to specify whether the path is an absolute or relative path. The valid values for this parameter are: • %FilePath_Relative (default) • %FilePath_Absolute If you don't specify *pathtype* the default is %FilePath_Relative. If the path is an absolute path, whatever path you specify is used verbatim. You must specify a drive letter as well as the complete path. You can't use any wildcards when specifying a path. The function automatically converts platform-specific separator characters to the appropriate form for where your PeopleCode program is executing. On a WIN32 system, UNIX "/" separators are converted to "\", and on a UNIX system, WIN32 "\" separators are converted to "/".

Note. The syntax of the file path does *not* depend on the file system of the platform where the file is actually stored; it depends only on the platform where the PeopleCode is executing.

Returns

A Boolean value: True if the file exists, False if it doesn't.

Example

The following example opens a file for appending if it exists on the system:

```
If FileExists("c:\work\item.txt", %FilePath_Absolute) Then
   &MYFILE = GetFile("c:\work\item.txt", "A");
   /* Process the file */
   &MYFILE.Close();
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, FindFiles , page 189, Chapter 1, PeopleCode Built-in Functions, GetFile , page 243, Chapter 13, File Class , page 873, Chapter 13, File Class , Open, page 882.

Find

Syntax

Find(*string*, *within_string* [, *number*])

Description

Find locates one string of text within another string of text and returns the character position of the string as an integer. Find is case-sensitive and does not allow wildcards.

If you need to do either case-sensitive search or pattern matching, just to find if a string matches a pattern, use the **DBPatternMatch** function.

Parameters

Parameter	Description
string	The text you are searching for.
within_string	The text string you are searching within.
number	The position of *within_string* at which you want to start your search. If you omit *number*, Find starts at the first character of *within_string*.

Returns

Returns a Number value indicating the starting position of *string* in *within_string*.

Find returns with 0 if *string* does not appear in *within_string*, if *number* is less than or equal to zero, or if *number* is greater than the length of *within_string*.

Example

In the following example, the first statement returns 1; the second statement returns 6.

```
&POS = Find("P", "PeopleSoft")
&POS = Find("e", "PeopleSoft", 4)
```

See Also

Chapter 1, PeopleCode Built-in Functions, Exact , page 170, Chapter 1, PeopleCode Built-in Functions, Len , page 313, Chapter 1, PeopleCode Built-in Functions, DBPatternMatch , page 129.

Findb

Syntax

Findb(*string*, *within_string* [, *number*])

Description

Findb locates one string of text within another string of text and returns the byte of the string as an integer. Findb is byte-oriented, whereas **Find** is character-oriented; this distinction is significant whenever the source string contains double-byte characters.

Findb is case-sensitive and does not allow wildcards.

Findb displays an error message if *string* does not appear in *within_string*, if *number* is less than zero, or if *number* is greater than the length of *within_string*.

Parameters

Parameter	Description
string	The text you are searching for.
within_string	The text string you are searching within.
number	The position of *within_string* at which you want to start your search. If you omit *number*, Findb starts at the first byte of *within_string*.

Returns

Returns a Number value indicating the starting position of *string* in *within_string*.

Example

In the following example, the first statement returns 1; the second statement returns 7.

```
&POS = Find("P", "PeopleSoft")
&POS = Find("S", "PeopleSoft", 4)
```

See Also

Chapter 1, PeopleCode Built-in Functions, Lenb , page 314.

FindCodeSetValues

Syntax

FindCodeSetValues(*CodesetName, &NameValuePairs, SourceNodeName, TargetNodeName*)

Description

Use the **FindCodeSetValues** function to find a list of code set name-value pairs. Code sets are primarily used with data value translations as part of a transformation.

Parameters

Parameter	Description
CodeSetName	Specify the name of the code set you want to find, as a string.
&NameValuePairs	Specify a 2 dimensional array containing the name value pairs in the specified code set that you want to use.
SourceNodeName	Specify the name of the source (initial) node used in the data transformation.
TargetNodeName	Specify the name of the target (result) node used in the data transformation.

Returns

A two-dimensional array of any.

Example

This example checks the specified CodeSet values, with the name value pairs of "locale/en_us" and "uom/box". It takes the returned array and adds XML nodes to the document. The XML nodes names are the unique names of the CodeSet value, and the XML node value is the corresponding return value.

```
/* Get the data from the AE Runtime */
Local TransformData &incomingData = %TransformData;

/* Set a temp object to contain the incoming document */
Local XmlDoc &tempDoc = &incomingData.XmlDoc;

/* Declare the node */
Local XmlNode &tempNode;

/* Create an array to hold the name value pairs */
   Local array of array of string &inNameValuePairsAry;

/* Clear out the doc and put in a root node */
If (&tempDoc.ParseXmlString("<?xml version=""1.0""?><xml/>")) Then

   /* Load the array with some values */
   &inNameValuePairsAry = CreateArray(CreateArray("locale", "en_us"), CreateArray⇒
("uom", "box"));

   /* Find the codeset values */
   &outAry = FindCodeSetValues("PS_SAP_PO_01", &inNameValuePairsAry, "SAP_SRC",⇒
 "PSFT_TGT");

/*    Local XmlNode &tempNode; */

   /* Make sure something was returned */
   If &outAry.Len > 0 Then

      /* Loop through the quantities and make sure they are all above 5 */
      For &i = 1 To &outAry.Len

         /* Add the current system date to the working storage*/
         &tempNode = &tempDoc.DocumentElement.AddElement(&outAry [&i][1]);
         &tempNode.NodeValue = &outAry [&i][2];

      End-For;
   End-If;
End-If;
```

See Also

Chapter 7, Array Class , page 633, *PeopleSoft Integration Broker*, Applying Filtering, Transformation, and Translation.

FindFiles

Syntax

```
FindFiles(filespec_pattern [, pathtype])
```

Description

The **FindFiles** function returns a list of the external filenames that match the filename pattern you provide, in the location you specify.

Working With Relative Paths

If you specify a relative path, that path is appended to the path specified by an environmental variable, if the variable exists, in a particular order. This means if the first environmental variable in order is found, that is the one that's used. If it isn't found, the next one is searched for, and if found, is used, and so on.

The environmental variables that are used are different, depending on whether the machine where the code is running is an application server. If the code is running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]

If the code is *not* running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]
3. TEMP[\relative_path]

The PS_FILEDIR environmental variable is *not* initialized and set to a path automatically. To use this environmental variable, you must set it yourself.

Note. Your system security should verify if a user has the correct permissions before allowing access to a drive (for example, if a user changes their **TEMP** environmental variable to a network drive they don't normally have access to, your system security should detect this.)

Parameters

Parameter	Description
filespec_pattern	Specify the path and filename pattern for the files you want to find. The path can be any string expression that represents a single relative or absolute directory location. The filename pattern, but not the path, can include two wildcards: ***** (Asterisk): matches zero or more characters at its position. **?** (Question mark): matches exactly one character at its position.
pathtype	If you have prepended a path to the file name, use this parameter to specify whether the path is an absolute or relative path. The valid values for this parameter are: • %FilePath_Relative (default) • %FilePath_Absolute If you don't specify *pathtype* the default is %FilePath_Relative. If the path is an absolute path, whatever path you specify is used verbatim. You must specify a drive letter as well as the complete path. You can't use any wildcards when specifying a path. The function automatically converts platform-specific separator characters to the appropriate form for where your PeopleCode program is executing. On a WIN32 system, UNIX "/" separators are converted to "\", and on a UNIX system, WIN32 "\" separators are converted to "/".

Note. The syntax of the file path does *not* depend on the file system of the platform where the file is actually stored; it depends only on the platform where the PeopleCode is executing.

Returns

A string array whose elements are filenames qualified with the same relative or absolute path you specified in the input parameter to the function.

Example

The following example finds all files in the system's TEMP location whose names end with ".txt", then opens and processes each one in turn:

```
Local array of string &FNAMES;
Local file &MYFILE;

&FNAMES = FindFiles("\*.txt");
while &FNAMES.Len > 0
   &MYFILE = GetFile(&FNAMES.Shift(), "R");  /* Open each file */
      /* Process the file contents */
   &MYFILE.Close();
end-while;
```

See Also

Chapter 1, PeopleCode Built-in Functions, FileExists , page 184, Chapter 1, PeopleCode Built-in Functions, GetFile , page 243, Chapter 13, File Class , page 873, Chapter 7, Array Class , page 633.

FlushBulkInserts

Syntax

FlushBulkInserts()

Description

The **FlushBulkInserts** function moves the bulk inserted rows from the bulk insert buffers of the PeopleSoft process to the physical tables on the database. This flushes all open SQL objects that have pending bulk inserts, but performs no COMMITs. If the flush fails, the PeopleCode program terminates.

When executing a SQL insert using a SQL object with the BulkMode property set to True, the rows being inserted cannot be selected by this database connection until the bulk insert is flushed. For another connection to the database to be able to select those rows, both a flush and a COMMIT are required. To have your process see the bulk inserted rows without committing and without closing the SQL object or its cursor (that is, maintaining reuse for the SQL object), use FlushBulkInserts.

An example of using this function would be in preparation for a database commit where you do not want to close the SQL insert statement, but need to ensure that all the rows you have inserted up to this point are in fact in the database and not in the buffer.

Another example would be when another SQL statement in the same PeopleSoft process needs to select rows that have been inserted using bulk insert and you do not want to close the SQL insert statement. The SELECT cannot read rows in the bulk insert buffer, so you need to flush them to the table from which the SELECT is reading.

Parameters

None.

Returns

None. If the flush fails, the PeopleCode program terminates.

Example

```
&CM_DEPLETION_REC = CreateRecord(Record.CM_DEPFIFO_VW);
&CM_DEPLETE_REC = CreateRecord(Record.CM_DEPLETE);
&DEPLETE_FIFO_SEL = GetSQL(SQL.CM_DEPLETE_FIFO_SEL);
&ONHAND_FIFO_SEL = GetSQL(SQL.CM_ONHAND_FIFO_SEL);
DEPLETE_INS = GetSQL(SQL.CM_DEPLETE_INS);
&DEPLETE_INS.BulkMode = True;

&DEPLETE_FIFO_SEL.Execute(&CM_DEPLETION_REC, CM_COSTING_AET.BUSINESS_UNIT, CM_⇒
COSTING_AET.CM_BOOK);
While &DEPLETE_FIFO_SEL.Fetch(&CM_DEPLETION_REC);
   /* Call functions that populate &CM_DEPLETE_REC.values */
   . . .
   &DEPLETE_INS.Execute(&CM_DEPLETE_REC);
   .  .  .
   If &CM_DEPLETION_REC.CM_COST_PROC_GROUP.Value = "BINTOBIN" Then
      /* Bin to Bin transfers are both a deplete and receipt, call functions to⇒
 create the receipt */
      .  .  .
```

```
            /* Flush Bulk Insert to be able to see the current on hand quantities in⇒
    CM_ONHAND_VW */
            FlushBulkInserts();
      End-if;
   End-While;
   .  .  .
```

See Also

PeopleSoft Integration Broker, Applying Filtering, Transformation, and Translation.

For

Syntax

```
For count = expression1 To expression2
    [Step i];
    statement_list
End-for
```

Description

The **For** loop causes the statements of the *statement_list* to be repeated until *count* is equal to *expression2*. **Step** specifies the value by which *count* will be incremented each iteration of the loop. If you do not include Step, *count* is incremented by 1 (or -1 if the start value is greater than the end value.) Any statement types are allowed in the loop, including other loops.

A **Break** statement inside the loop causes execution to continue with whatever follows the loop. If the Break occurs in a nested loop, the Break does not apply to the outside loop.

Example

The following example loops through all of the rows for the FIELDNAME scroll area:

```
&FIELD_CNT = ActiveRowCount(DBFIELD_VW.FIELDNAME);
For &I = 1 to &FIELD_CNT;
   WinMessage(MsgGetText(21000, 1, "Present Row Number is: %1", &I));
End-for;
```

FormatDateTime

Syntax

```
FormatDateTime(datetime, {timezone | "Local" | "Base"}, displayTZ)
```

Description

FormatDateTime takes a *datetime* value and converts it to text. If a specific time zone abbreviation, or a field reference, is passed in *timezone*, **FormatDateTime** adjusts the datetime to the user's local time zone instead of the specified time zone. The system's base time zone is specified on the PSOPTIONS table. The value *datetime* is assumed to be in base time.

See *PeopleSoft Server Tools Administration*, PeopleTools Utilities.

If **Local** is specified, FormatDateTime adjusts the datetime to the user's local time zone instead of a specific time zone.

If **True** is specified for *displayTZ* FormatDateTime appends the time zone abbreviation to the returned string.

Parameters

Parameter	Description
datetime	Specify the datetime value to be formatted.
timezone | **Local** | **Base**	Specify a value for converting *datetime*. The values are: • *timezone* - a time zone abbreviation or a field reference to be used for converting *textdatetime*. • **Local** - use the local time zone for converting *textdatetime*. • **Base** - use the base time zone for converting *textdatetime*.
displayTZ	Specify whether the time zone abbreviation should be appended to the returned string. This parameter takes a Boolean: True if the abbreviation should be appended, False, otherwise.

Returns

A formatted string value.

Example

The following example populates the &DISPDATE variable with a string containing the datetime value in the ORDER_DATE field adjusted to the user's local time zone, and with the time zone abbreviation.

```
&DISPDATE=FormatDateTime(ORDER_DATE, "Local", True);
```

The following example populates the &DISPDATE variable with a string containing the datetime value in the SHIP_DATE field adjusted to the time zone stored in the SHIP_TZ field, and does not include the time zone abbreviation in the output.

```
&DISPDATE=FormatDateTime(SHIP_DATE, SHIP_TZ, False);
```

See Also

Chapter 1, PeopleCode Built-in Functions, ConvertDatetimeToBase , page 85, Chapter 1, PeopleCode Built-in Functions, ConvertTimeToBase , page 87, Chapter 1, PeopleCode Built-in Functions, DateTimeToLocalizedString , page 121, Chapter 1, PeopleCode Built-in Functions, IsDaylightSavings , page 302, Chapter 1, PeopleCode Built-in Functions, DateTimeToTimeZone , page 124, Chapter 1, PeopleCode Built-in Functions, TimeToTimeZone , page 460, Chapter 1, PeopleCode Built-in Functions, TimeZoneOffset , page 461, Chapter 1, PeopleCode Built-in Functions, DateTimeToHTTP , page 120.

Forward

Syntax

Forward(*from physical queue ID, from agent ID, task number, task type, to logical queue ID*[*, to agent ID*])

Description

Forward transfers a task from one agent to another agent or from one agent's logical queue to another logical queue. This enables agents to reroute tasks that are not appropriate for their skill level or functional expertise. Keep the following in mind when using **Forward**:

- The queue server subtracts the task's cost from the transferring agent's workload.
- The system cannot forward tasks to logical queues that do not have active physical queues on the same MultiChannel Framework cluster as the physical queue to which the task currently belongs. That is, you can't forward tasks *across* MultiChannel Framework clusters.
- A queue server does not allow a task to be transferred if the agent who owns that task is not logged on to that queue server. PeopleSoft recommends that you only use **Forward** for application pages that the MultiChannel Framework console launches when agents accept or activate assigned tasks.
- **Forward** only applies to email and generic task types.

Parameters

Parameter	Description
from physical queue ID	The physical queue is the internal representation of the logical queue that the agent signs onto and to which the task currently belongs. This is a string value, such as "sales3" or "marketing2." You retrieve the current physical queue from the query string in the URL of the page launched by the MultiChannel Framework console. Use the Request class **GetParameter** method with the value `ps_qid`
from agent ID	Specifies the current agent, as in the agent that "accepted" the task. This is a string value. You retrieve the current physical queue from the query string in the URL of the page launched by the MultiChannel Framework console. Use the Request class **GetParameter** method with the value `ps_agentid`.
task number	Identifies the task to be forwarded. **EnQueue** returns this value. This is a string value. You retrieve the current physical queue from the query string in the URL of the page launched by the MultiChannel Framework console. Use the Request class **GetParameter** method with the value `ps_tasknum`.
task type	Identifies the task type. This value is provided by the queue. This is a string value. Valid values are: • email • generic You retrieve the current physical queue from the query string in the URL of the page launched by the MultiChannel Framework console. Use the Request class **GetParameter** method with the value `ps_tasktype`.
to logical queue ID	Specifies the logical queue to which the system forwards the task. This is a string value. The queue ID is case sensitive and must match the case used when you created the queue using the Queues page.
to agent ID	This is an optional parameter. It is a string value specifying a particular agent ID to receive the forwarded task. If this value is specified, the system holds the task until the specified agent is available on the new queue to take this task. This means that the specified agent must be able to log in to one of the physical queues belonging to the destination logical queue. The system determines which physical queue the specified agent has access to and assigns the task to that queue for that agent. If the agent ID is not specified, the physical queue is chosen at random from the active physical queues. **Note.** For better performance, PeopleSoft recommends not specifying the target agent as this has a processing overhead for the queue servers and does not allow the system to balance workload across all available agents.

Returns

Returns 0 on success.

If unsuccessful, it returns a message number. The message set ID for MultiChannel Framework is 162.

For example, 1302 is returned when an invalid task type or no value is provided.

Example

```
Forward("SALES5", "TSAWYER", "email_2145", "email", "MARKETING", "GSALMON");
```

The following example shows how to retrieve parameters from the application page using the Request class **GetParameter** method.

```
PSMCFFUNCLIB.MCF_QUEUE.Value = %Request.GetParameter("ps_qid");
PSMCFFUNCLIB.MCF_TASKTYPE.Value = %Request.GetParameter("ps_tasktype");
PSMCFFUNCLIB.MCF_TASKNUM.Value = %Request.GetParameter("ps_tasknum");
PSMCFFUNCLIB.MCF_AGENTID.Value = %Request.GetParameter("ps_agentid");

&nret = Forward(PSMCFFUNCLIB.MCF_QUEUE, PSMCFFUNCLIB.MCF_AGENTID, PSMCFFUNCLIB.⇒
MCF_TASKNUM, PSMCFFUNCLIB.MCF_TASKTYPE, &ToQueue);

If &nret = 0 Then
      MessageBox(0, "", 0, 0, "Successfully forwarded.");
      End-If
```

Function

Syntax

Function *name* [(*paramlist)*] [**Returns** *data_type*]
   [*statements*]
End-function

Where *paramlist* is:
&param1 [**As** *data_type*] [**,** *&param2* [**As** *data_type*]]...

Where *data_type* is any valid data type, including Number, String, Date, Rowset, SQL, Record, and so on.

Where *statements* is a list of PeopleCode statements.

Description

PeopleCode functions can be defined in any PeopleCode program. Function definitions must be placed at the top of the program, along with any variable and external function declarations.

Functions can be called from the program in which they are defined, in which case they don't need to be declared, and they can be called from another program, in which case they need to be declared at the top of the program where they are called.

Any variables declared within a function are valid for the scope of the function.

By convention, external PeopleCode functions are stored in records whose names begin in FUNCLIB_, and they are always placed in the FieldFormula event (which is convenient because this event should no longer be used for anything else).

Note. Functions can be stored in the FieldFormula event only for record fields, *not* for component record fields.

A function definition consists of:

- The keyword **Function** followed by the name of the function and an optional list of parameters. The name of the function can be up to 100 characters in length.
- An optional **Returns** clause specifying the data type of the value returned by the function.
- The statements to be executed when the function is called.
- The **End-function** keyword.

The parameter list, which must be enclosed in parentheses, is a comma-separated list of variable names, each prefixed with the & character. Each parameter is optionally followed by the keyword **As** and the name for one of the conventional PeopleCode data types (Number, String, Date, and so on) or any of the object data types (such as Rowset, SQL, and so on.) If you specify data types for parameters, then function calls are checked to ensure that values passed to the function are of the appropriate type. If data types are not specified, then the parameters, like other temporary variables in PeopleCode, take on the type of the value that is passed to them.

PeopleCode parameters are always passed by reference. This means that if you pass the function a variable from the calling routine and change the value of the variable within the function, the value of the variable is changed when the flow of execution returns to the calling routine.

If the function is to return a result to the caller, the optional **Returns** part must be included to specify the data type of the returned value. You have seven choices of value types: Number, String, Date, Time, DateTime, Boolean, or Any.

PeopleCode internal subroutines are part of the enclosing program and can access the same set of variables as the other statement-lists of the program, in addition to local variables created by the parameters and local variable declarations within the function.

Returning a Value

You can optionally return a value from a PeopleCode function. To do so, you must include a **Return** statement in the function definition, as described in the preceding section. For example, the following statement returns a Number value:

```
function calc_something(&parm1 as number, &parm2 as number) returns number
```

In the code section of your function, use the **Return** statement to return a value to the calling routine. When the **Return** statement executes, the function ends and the flow of execution goes back to the calling routine.

Example

This example returns a Boolean value based on the return value of a **SQLExec**:

```
function run_status_upd(&PROCESS_INSTANCE, &RUN_STATUS) returns Boolean;
   &UPDATEOK = SQLExec("update PS_PRCS_RQST set run_status = :1  ⇒
where process_instance = :2", &RUN_STATUS, &PROCESS_INSTANCE);
   If &UPDATEOK Then
      Return True;
   Else
      Return False;
   End-if;
End-function;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Declare Function , page 130, Chapter 1, PeopleCode Built-in Functions, Return , page 376.

GenerateActGuideContentUrl

Syntax

```
GenerateActGuideContentlUrl(PORTAL.portalname, NODE.nodename, MENUNAME.menuname,
Marketname, COMPONENT.componentname, ActivityGuide)
```

Description

The **GenerateActGuideContentUrl** function creates a URL string that represents an absolute reference to the specified activity guide (life event) for the content servlet. The ContentURI of the node that hosts the specified portal is used in the generated URL. The URL contains a reference to the content service (*psc*) servlet.

If you want to generate a URL for the portal service (*psp*) servlet, use **GenerateActGuidePortalURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the activity guide, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
menuname	Specify the name of the menu containing the activity guide, prefixed with the reserved word **MENUNAME**. You can also use a string, such as %Menu, for this value.
Marketname	Specify the name of the market of the component. You can also use a string, such as %Market, for this value.
ComponentName	Specify the name of the component, prefixed with the reserved word **COMPONENT**. You can also use a string, such as %Component, for this value.
ActivityGuide	Specify the name of the Activity Guide, as a string.

Returns

A string with the following format:

```
http://Content URI of node/portal/node/l/ActivityGuide.component.market
```

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&AGURL = GenerateActGuideContentUrl(%Portal, %Node, MENUNAME.MAINTAIN_SECURITY,⇒
 "GBL", COMPONENT.CHANGE_PASSWORD, "QE_ACTIVITY_GUIDE_DEMO");
```

might produce the following URL string:

```
http://boynten8700/psc/ps/EMPLOYEE/QE_LOCAL/l/QE_ACTIVITY_GUIDE_DEMO.MAINTAIN_⇒
SECURITY.CHANGE_PASSWORD.GBL
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateActGuidePortalUrl , page 199, Chapter 1, PeopleCode Built-in Functions, GenerateActGuideRelativeUrl , page 200, *PeopleSoft Workflow*, Designing Activity Guides.

GenerateActGuidePortalUrl

Syntax

GenerateActGuidePortalUrl(**PORTAL**.*portalname*, **NODE**.*nodename*, **MENUNAME**.*menuname*, *Marketname*, **COMPONENT**.*componentname*, *ActivityGuide*)

Description

The **GenerateActGuidePortalUrl** function creates a URL string that represents an absolute reference to the specified activity guide (life event) for the portal servlet. The PortalURI of the node that hosts the specified portal is used in the generated URL. The URL contains a reference to the portal service (*psp*) servlet.

If you want to generate a URL for the portal content (*psc*) servlet, use **GenerateActGuideContentURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the activity guide, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
menuname	Specify the name of the menu containing the activity guide, prefixed with the reserved word **MENUNAME**. You can also use a string, such as %Menu, for this value.
Marketname	Specify the name of the market of the component. You can also use a string, such as %Market, for this value.
ComponentName	Specify the name of the component, prefixed with the reserved word **COMPONENT**. You can also use a string, such as %Component, for this value.
ActivityGuide	Specify the name of the Activity Guide, as a string.

Returns

A string with the following format:

http://*Portal URI of node/portal/node/***l**/*ActivityGuide.component.market*

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&AGURL = GenerateActGuidePortalUrl(%Portal, %Node, MENUNAME.MAINTAIN_SECURITY,⇒
 "GBL", COMPONENT.CHANGE_PASSWORD, "QE_ACTIVITY_GUIDE_DEMO");
```

might create the following URL string:

```
http://boynte700/psp/ps/EMPLOYEE/QE_LOCAL/l/QE_ACTIVITY_GUIDE_DEMO.MAINTAIN_⇒
SECURITY.CHANGE_PASSWORD.GBL
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateActGuideContentUrl , page 198, Chapter 1, PeopleCode Built-in Functions, GenerateActGuideRelativeUrl , page 200, *PeopleSoft Workflow*, Designing Activity Guides.

GenerateActGuideRelativeUrl

Syntax

GenerateActGuideRelativeUrl(**PORTAL**.*portalname*, **NODE**.*nodename*, **MENUNAME**.*menuname*, M*arketname*, **COMPONENT**.*componentname*, *ActivityGuide*)

Description

The **GenerateActGuideContentUrl** function creates a URL string that represents an relative reference to the specified activity guide (life event). The relative reference is suitable for use on any page that itself has the simple URL format.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the activity guide, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
menuname	Specify the name of the menu containing the activity guide, prefixed with the reserved word **MENUNAME**. You can also use a string, such as %Menu, for this value.
Marketname	Specify the name of the market of the component. You can also use a string, such as %Market, for this value.
ComponentName	Specify the name of the component, prefixed with the reserved word **COMPONENT**. You can also use a string, such as %Component, for this value.
ActivityGuide	Specify the name of the Activity Guide, as a string.

Returns

A string with the following format:

```
../../../Portal/node/l/ActivityGuide.menu.component.market
```

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&AGURL = GenerateActGuideRelativeUrl(%Portal, %Node, MENUNAME.MAINTAIN_SECURITY,⇒
 "GBL", COMPONENT.CHANGE_PASSWORD, "QE_ACTIVITY_GUIDE_DEMO");
```

might produce the following URL string:

```
../../../EMPLOYEE/QE_LOCAL/l/QE_ACTIVITY_GUIDE_DEMO.MAINTAIN_SECURITY.CHANGE_⇒
PASSWORD.GBL
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateActGuideContentUrl , page 198, Chapter 1, PeopleCode Built-in Functions, GenerateActGuidePortalUrl , page 199, *PeopleSoft Workflow*, Designing Activity Guides.

GenerateComponentContentRelURL

Syntax

GenerateComponentContentRelURL(**PORTAL**.*portalname*, **NODE**.*nodename*, **MENUNAME**.*menuname*, **MARKET**.*marketname*, **COMPONENT**.*componentname*, **PAGE**.*pagename*, *action*, [, *keylist*])

where *keylist* is a list of field references in the form:
[*recordname*.]*field1* [, [*recordname*.]*field2*]. . .

OR
&RecordObject1 [, *&RecordObject2*]. . .

Description

The **GenerateComponentContentRelURL** function creates a URL string that represents a relative reference to the specified component for the content servlet. The relative reference is suitable for use on any page that itself has the simple URL format.

If you want to generate an absolute URL for a component, use the **GenerateComponentContentURL** function.

Parameters

Parameter	Description
PortalName	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value. This parameter is ignored by the content service, but is a required part of the `psc` URL format.
NodeName	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
MenuName	Specify the name of the menu containing the content, prefixed with the reserved word **MENUNAME**. You can also use a string, such as %Menu, for this value.
Marketname	Specify the name of the market of the component, prefixed with the reserved word **MARKET**. You can also use a string, such as %Market, for this value.
ComponentName	Specify the name of the component, prefixed with the reserved word **COMPONENT**. You can also use a string, such as %Component, for this value.
Pagename	Specify the name of the page that contains the content. If you specify a page name, it must be prefixed with the keyword **PAGE**. You can also specify an empty string ("") for this value.
Action	Specify a single-character code. Valid actions are: • "A" (add) • "U" (update) • "L" (update/display all) • "C" (correction) • "E" (data entry) You can also specify an empty string ("") for this value.
Keylist	An optional list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. If a record object is specified, any field of that record object *that is also a field of the search record for the destination component* is added to keylist. The keys in the *fieldlist* must uniquely identify a row in the "to" page search record. If a unique row is not identified, of if Force Search Processing has been selected, the search dialog appears. If the *keylist* parameter is not supplied the destination component's search key must be found as part of the source components level 0 record buffer.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

```
../../../Portal/node/c/menu.component.market?parameters
```

If the node has a Node Type of ICType, a string of the following format is returned:

```
../../../portal/node/?ICType=Panel&Menu=menu&Market=market&PanelGroupName=component⇒
?parameters
```

The question mark and the text following the question mark may or may not be included, depending on whether or not you specified a page and action or not.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&MyCompURL = GenerateComponentContentRelURL("EMPLOYEEPORTAL", "CRM", MenuName.SFA,⇒
 "GBL", Component.CUSTOMERINFO, Page.CUST_DATA1, "U", EMPLID);
```

Might create the following URL:

```
../../../psc/PS84/EMPLOYEEPORTAL/CRM/c/SFA.CUSTOMERINFO.GBL?page=CUST_⇒
DATA1&&Action=U&emplid=00001
```

Because this function terminates if the portal or node name is invalid, it's enclosed in a try-catch section so if an exception gets raised, it can be handled.

```
try
   &MyURL = GenerateComponentContentRelURL(%Portal, "HRMS", Menuname.ADMIN_⇒
WORKFORCE, "GBL", Component.ABSENCE_HISTORY, Page. ABSENCE_HISTORY, "U", EMPLID)

   catch ExceptionPortal &Ex1
      /* error handling portal name not valid */
   catch ExceptionNode &Ex2
      /* error handling Node name not valid */

end-try;
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentURL , page 203, Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalRelURL , page 205, Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalURL , page 207, Chapter 1, PeopleCode Built-in Functions, GenerateComponentRelativeURL , page 209, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateComponentContentURL

Syntax

GenerateComponentContentURL(**PORTAL**.*portalname*, **NODE**.*nodename*, **MENUNAME**.*menuname*, **MARKET**.*marketname*, **COMPONENT**.*componentname*, **PAGE**.*pagename*, *action*, [, *keylist*])

where *keylist* is a list of field references in the form:
[*recordname*.]*field1* [, [*recordname*.]*field2*]. . .

OR
&RecordObject1 [, *&RecordObject2*]. . .

Description

The **GenerateComponentContentURL** function creates a URL string that represents an absolute reference to the specified component for the content servlet.

The ContentURI of the specified node is used in the generated URL. The URL contains a reference to the portal content (*psc*) servlet. If you want to generate a URL for the portal service (*psp*), use **GenerateComponentPortalURL**.

Parameters

Parameter	Description
PortalName	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value. This parameter is ignored by the content service, but is a required part of the `psc` URL format.
NodeName	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
MenuName	Specify the name of the menu containing the content, prefixed with the reserved word **MENUNAME**. You can also use a string, such as %Menu, for this value.
Marketname	Specify the name of the market of the component, prefixed with the reserved word **MARKET**. You can also use a string, such as %Market, for this value.
ComponentName	Specify the name of the component, prefixed with the reserved word **COMPONENT**. You can also use a string, such as %Component, for this value.
Pagename	Specify the name of the page that contains the content. If you specify a page name, it must be prefixed with the keyword **PAGE**. You can also specify an empty string ("") for this value.
Action	Specify a single-character code. Valid actions are: • "A" (add) • "U" (update) • "L" (update/display all) • "C" (correction) • "E" (data entry) You can also specify an empty string ("") for this value.
Keylist	An optional list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. If a record object is specified, any field of that record object *that is also a field of the search record for the destination component* is added to keylist. The keys in the *fieldlist* must uniquely identify a row in the "to" page search record. If a unique row is not identified, of if Force Search Processing has been selected, the search dialog appears. If the *keylist* parameter is not supplied the destination component's search key must be found as part of the source components level 0 record buffer.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

```
http://Content URI of host node/Portal/node/c/menu.component.market?parameters
```

If the node has a Node Type of ICType, a string of the following format is returned:

```
http://Content URI of host node/portal/node/?ICType=Panel&Menu=menu&Market=market⇒
&PanelGroupName=component?parameters
```

The question mark and the text following the question mark may or may not be included, depending on whether or not you specified a page and action or not.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&MyCompURL = GenerateComponentContentURL("EMPLOYEEPORTAL", "CRM", MenuName.SFA,⇒
 "GBL", Component.CUSTOMERINFO, Page.CUST_DATA1, "U", EMPLID);
```

Might create the following URL:

```
http://serverx/servlets/psc/PS84/EMPLOYEEPORTAL/CRM/c/SFA.CUSTOMERINFO.GBL?page=⇒
CUST_DATA1&&Action=U&emplid=00001
```

Because this function terminates if the portal or node name is invalid, it's enclosed in a try-catch section so if an exception gets raised, it can be handled.

```
try
   &MyURL = GenerateComponentContentURL(%Portal, "HRMS", Menuname.ADMIN_WORKFORCE,⇒
 "GBL", Component.ABSENCE_HISTORY, Page. ABSENCE_HISTORY, "U", EMPLID)

   catch ExceptionPortal &Ex1
      /* error handling portal name not valid */
   catch ExceptionNode &Ex2
      /* error handling Node name not valid */

end-try;
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentRelURL , page 201, Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalRelURL , page 205, Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalURL , page 207, Chapter 1, PeopleCode Built-in Functions, GenerateComponentRelativeURL , page 209, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateComponentPortalRelURL

Syntax

GenerateComponentPortalRelURL(**PORTAL**.*portalname*, **NODE**.*nodename*, **MENUNAME**.*menuname*, **MARKET**.*marketname*, **COMPONENT**.*componentname*, **PAGE**.*pagename*, *action*, [, *keylist*])

where *keylist* is a list of field references in the form:
[*recordname*.]*field1* [, [*recordname*.]*field2*]. . .

OR
&RecordObject1 [, *&RecordObject2*]. . .

Description

The **GenerateComponentPortalRelURL** function creates a URL string URL string that represents a relative reference the specified content (component). The relative reference is suitable for use on any page that itself has the simple URL format.

If you want to generate an absolute URL for a component, use the **GenerateComponentPortalURL** function.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
menuname	Specify the name of the menu containing the content, prefixed with the reserved word **MENUNAME**. You can also use a string, such as %Menu, for this value.
Marketname	Specify the name of the market of the component, prefixed with the reserved word **MARKET**. You can also use a string, such as %Market, for this value.
ComponentName	Specify the name of the component, prefixed with the reserved word **COMPONENT**. You can also use a string, such as %Component, for this value.
pagename	Specify the name of the page that contains the content. If you specify a page name, it must be prefixed with the keyword **PAGE**. You can also specify a Null string ("") for this value.
action	Specify a single-character code. Valid actions are: • "A" (add) • "U" (update) • "L" (update/display all) • "C" (correction) • "E" (data entry) You can also specify a Null string ("") for this value.
keylist	An optional list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. If a record object is specified, any field of that record object *that is also a field of the search record for the destination component* is added to keylist. The keys in the *fieldlist* must uniquely identify a row in the "to" page search record. If a unique row is not identified, of if Force Search Processing has been selected, the search dialog appears. If the *keylist* parameter is not supplied the destination component's search key must be found as part of the source components level 0 record buffer.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

```
../../../portal/node/c/menu.component.market?parameters
```

If the node has a Node Type of ICType, a string of the following format is returned:

```
../../../portal/node/?ICType=Panel&Menu=menu&Market=market&PanelGroupName=component⇒
?parameters
```

The question mark and the text following the question mark may or may not be included, depending on whether or not you specified a page and action or not.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&MyCompURL = GenerateComponentPortalRelURL("EMPLOYEEPORTAL", "CRM", MenuName.SFA,⇒
 "GBL", Component.CUSTOMERINFO, , "", "");
```

Might create the following URL:

```
../../../EMPLOYEEPORTAL/CRM/c/sfa.customerinfo.gbl
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentRelURL , page 201, Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentURL , page 203, Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalURL , page 207, Chapter 1, PeopleCode Built-in Functions, GenerateComponentRelativeURL , page 209, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateComponentPortalURL

Syntax

GenerateComponentPortalURL(**PORTAL**.*portalname*, **NODE**.*nodename*, **MENUNAME**.*menuname*, **MARKET**.*marketname*, **COMPONENT**.*componentname*, **PAGE**.*pagename*, *action*, [, *keylist*])

where *keylist* is a list of field references in the form:
[*recordname*.]*field1* [, [*recordname*.]*field2*]. . .

OR
&RecordObject1 [, *&RecordObject2*]. . .

Description

The **GenerateComponentPortalURL** function creates a URL string that represents an absolute reference to the specified component for the portal servlet. The PortalURI of the node that hosts the specified portal is used in the generated URL. The URL contains a reference to the portal service (*psp*) servlet.

If you want to generate a URL for the portal content (*psc*) servlet, use **GenerateComponentContentURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
menuname	Specify the name of the menu containing the content, prefixed with the reserved word **MENUNAME**. You can also use a string, such as %Menu, for this value.
Marketname	Specify the name of the market of the component, prefixed with the reserved word **MARKET**. You can also use a string, such as %Market, for this value.
ComponentName	Specify the name of the component, prefixed with the reserved word **COMPONENT**. You can also use a string, such as %Component, for this value.
pagename	Specify the name of the page that contains the content. If you specify a page name, it must be prefixed with the keyword **PAGE**. You can also specify a Null string ("") for this value.
action	Specify a single-character code. Valid actions are: • "A" (add) • "U" (update) • "L" (update/display all) • "C" (correction) • "E" (data entry) You can also specify a Null string ("") for this value.
keylist	An optional list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. If a record object is specified, any field of that record object *that is also a field of the search record for the destination component* is added to keylist. The keys in the *fieldlist* must uniquely identify a row in the "to" page search record. If a unique row is not identified, of if Force Search Processing has been selected, the search dialog appears. If the *keylist* parameter is not supplied the destination component's search key must be found as part of the source components level 0 record buffer.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

```
http://Portal URI of host node/portal/node/c/menu.component.market?parameters
```

If the node has a Node Type of ICType, a string of the following format is returned:

```
http://Portal URI of host node/portal/node/?ICType=Panel&Menu=menu&Market=market⇒

&PanelGroupName=component?parameters
```

The question mark and the text following the question mark may or may not be included, depending on whether or not you specified a page and action or not.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&MyCompURL = GenerateComponentPortalURL("EMPLOYEEPORTAL", "CRM", MenuName.SFA,⇒
 "GBL", Component.CUSTOMERINFO, , "", "");
```

Might create the following URL:

```
http://mike.com/servlets/ps<b>p</b>/testsite/EMPLOYEEPORTAL/CRM/c/sfa.customerinfo.gbl
```

The following example uses a de-referenced name for the component.

```
&sComponent = "Component." | &sComponent;
&sPage = "Page.EM_VCHR_PYMNT_CLN";

&rwCurrent = GetRow();
/*- The Search Record keys -*/
&sQueryString = &sQueryString | "&BUSINESS_UNIT=" | &rwCurrent.EM_VCHR_INQ_VW.EM_⇒
BUSINESS_UNIT.Value;
&sQueryString = &sQueryString | "&VOUCHER_ID=" | &rwCurrent.EM_VCHR_INQ_VW.VOUCHER_⇒
ID.Value;

&sQueryString = GenerateComponentPortalURL(%Portal, %Node, MenuName.EM_BILL_⇒
PRESENTMENT, %Market, @&sComponent, @&sPage, "U") | &sQueryString;

%Response.RedirectURL(&sURL);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentRelURL , page 201, Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentURL , page 203, Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalRelURL , page 205, Chapter 1, PeopleCode Built-in Functions, GenerateComponentRelativeURL , page 209, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateComponentRelativeURL

Syntax

GenerateComponentRelativeURL(**PORTAL**.*portalname*, **NODE**.*nodename*, **MENUNAME**.*menuname*, **MARKET**.*marketname*, **COMPONENT**.*componentname*, **PAGE**.*pagename*, *action*, [, *keylist*])

where *keylist* is a list of field references in the form:
[*recordname.*]*field1* [, [*recordname*.]*field2*]. . .

OR
&RecordObject1 [, *&RecordObject2*]. . .

Description

The **GenerateComponentRelativeURL** function creates a URL string that represents a relative reference the specified content (component). The relative reference is suitable for use on any page that itself has the simple URL format.

If you want to generate an absolute URL for a component, use either **GenerateComponentContentURL** or **GenerateComponentPortalURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
menuname	Specify the name of the menu containing the content, prefixed with the reserved word **MENUNAME**. You can also use a string, such as %Menu, for this value.
Marketname	Specify the name of the market of the component, prefixed with the reserved word **MARKET**. You can also use a string, such as %Market, for this value.
ComponentName	Specify the name of the component, prefixed with the reserved word **COMPONENT**. You can also use a string, such as %Component, for this value.
Pagename	Specify the name of the page that contains the content. If you specify a page name, it must be prefixed with the keyword **PAGE**. You can also specify a Null string ("") for this value.
Action	Specify a single-character code. Valid actions are: • "A" (add) • "U" (update) • "L" (update/display all) • "C" (correction) • "E" (data entry) You can also specify a Null string ("") for this value.
Keylist	An optional list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. If a record object is specified, any field of that record object *that is also a field of the search record for the destination component* is added to keylist. The keys in the *fieldlist* must uniquely identify a row in the "to" page search record. If a unique row is not identified, of if Force Search Processing has been selected, the search dialog appears. If the *keylist* parameter is not supplied the destination component's search key must be found as part of the source components level 0 record buffer.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

```
../../../portal/node/c/menu.component.market?parameters
```

If the node has a Node Type of ICType, a string of the following format is returned:

```
../../../portal/node/?ICType=Panel&Menu=menu&Market=market&PanelGroupName=component⇒
?parameters
```

The question mark and the text following the question mark may or may not be included, depending on whether or not you specified a page and action or not.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code example:

```
&MyCompURL = GenerateComponentRelativeURL("EMPLOYEEPORTAL", "CRM", MenuName.SFA,⇒
 "GBL", Component.CUSTOMERINFO, "", "");
```

Might yield the following:

```
../../../EMPLOYEEPORTAL/CRM/c/sfa.customerinfo.gbl
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentRelURL , page 201, Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentURL , page 203, Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalRelURL , page 205, Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalURL , page 207, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateExternalPortalURL

Syntax

GenerateExternalPortalURL(**PORTAL**.*portalname*, **NODE**.*nodename*, *URL*)

Description

The **GenerateExternalPortalURL** function creates a URL string that represents an absolute reference the specified external content (URL) on the portal servlet.

The PortalURI of the node that hosts the specified portal is used in the generated URL. The generated URL contains a reference to the portal service (*psp*) servlet.

If you want to generate a relative URL, use **GenerateExternalRelativeURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
NodeName	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
URL	Specify the URL to be used for this content.

Returns

A string of the following format is returned:

```
http://Portal URI of host node/Portal/node/e/encodedURL
```

When the portal servlet evaluates an external URL, the Node is ignored, so %Node can always be passed in for the Node parameter.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&url = GenerateExternalPortalURL("EMPLOYEEPORTAL", "CRM", "http://www.excite.com");
```

Might create the following URL:

```
http://myserver/psp/ps/EMPLOYEEPORTAL/CRM/e/http%3a%2f%2fwww.excite.com
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateExternalRelativeURL , page 212, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateExternalRelativeURL

Syntax

```
GenerateExternalRelativeURL(PORTAL.portalname, NODE.nodename, EncodedURL)
```

Description

The **GenerateExternalRelativeURL** function creates a URL string that represents a relative reference the specified external content (URL). The relative reference is suitable for use on any page that itself has the simple URL format and which is served by the portal servlet (psp).

If you want to generate an absolute URL, use **GenerateExternalPortalURL**.

Parameters

Parameter	Description
PortalName	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
NodeName	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
EncodedURL	Specify the URL to be used for this content.

Returns

A string of the following format is returned:

```
../../../Portal/node/e/encodedURL
```

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&url = GenerateExternalRelativeURL("EMPLOYEEPORTAL", "CRM", "http:⇒
//www.excite.com");
```

Might create the following URL:

```
../../../EMPLOYEEPORTAL/CRM/e/http%3a%2f%2fwww.excite.com
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateExternalRelativeURL , page 212, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateHomepagePortalURL

Syntax

GenerateHomepagePortalURL(**PORTAL**.*portalname*, **NODE**.*nodename*, *Tabname*)

Description

The **GenerateHomepagePortalURL** function creates a URL string that represents an absolute reference the specified homepage tab on the portal servlet.

The PortalURI of the node that hosts the specified portal is used in the generated URL. The generated URL contains a reference to the portal service (*psp*) servlet.

If you want to generate a relative URL, use **GenerateHomepageRelativeURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value. **Note.** The value specified for this parameter is ignored. The node name that is used is automatically calculated. However, you must still specify a value for this parameter.
nodename	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value. This should be the node that hosts the specified portal.
Tabname	Specify the name of the tab on the homepage that you want to generate a URL for. If you specify a null string (""), the default tab is used.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

```
http://Portal URI of host node/Portal/node/h/?tab=tabname
```

This function returns a Null string if you specify an invalid portal or node.

Example

Specifying the following code:

```
&HomePage = GenerateHomepagePortalURL(%Portal, NODE.North_Asia, "");
```

Might generate the following string:

```
http://bejing/psp/psoft/crm/North_Asia/h/?tab=DEFAULT
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateHomepageRelativeURL , page 214, Chapter 21, PortalRegistry Classes , page 1093, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateHomepageRelativeURL

Syntax

```
GenerateHomepageRelativeURL(PORTAL.portalname, NODE.nodename, Tabname)
```

Description

The **GenerateHomepageRelativeURL** function creates a URL string that represents a relative reference the specified homepage on the portal servlet. The relative reference is suitable for use on any page that itself has the simple URL format.

If you want to generate an absolute URL, use **GenerateHomepagePortalURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value. . This should be the node that hosts the specified portal.
Tabname	Specify the name of the tab on the homepage that you want to generate a URL for. If you specify a null string (""), the default tab is used.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

```
../../../Portal/node/h/?tab=tabname
```

If the node has a Node Type of ICType, a string of the following format is returned:

```
./?cmd=start
```

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&HomePage = GenerateHomepageRelativeURL(%Portal, NODE.North_Asia, "");
```

Might generate the following string:

```
../../../crm/North_Asia/h/?tab=DEFAULT
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateHomepagePortalURL , page 213, Chapter 21, PortalRegistry Classes , page 1093, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateMobileTree

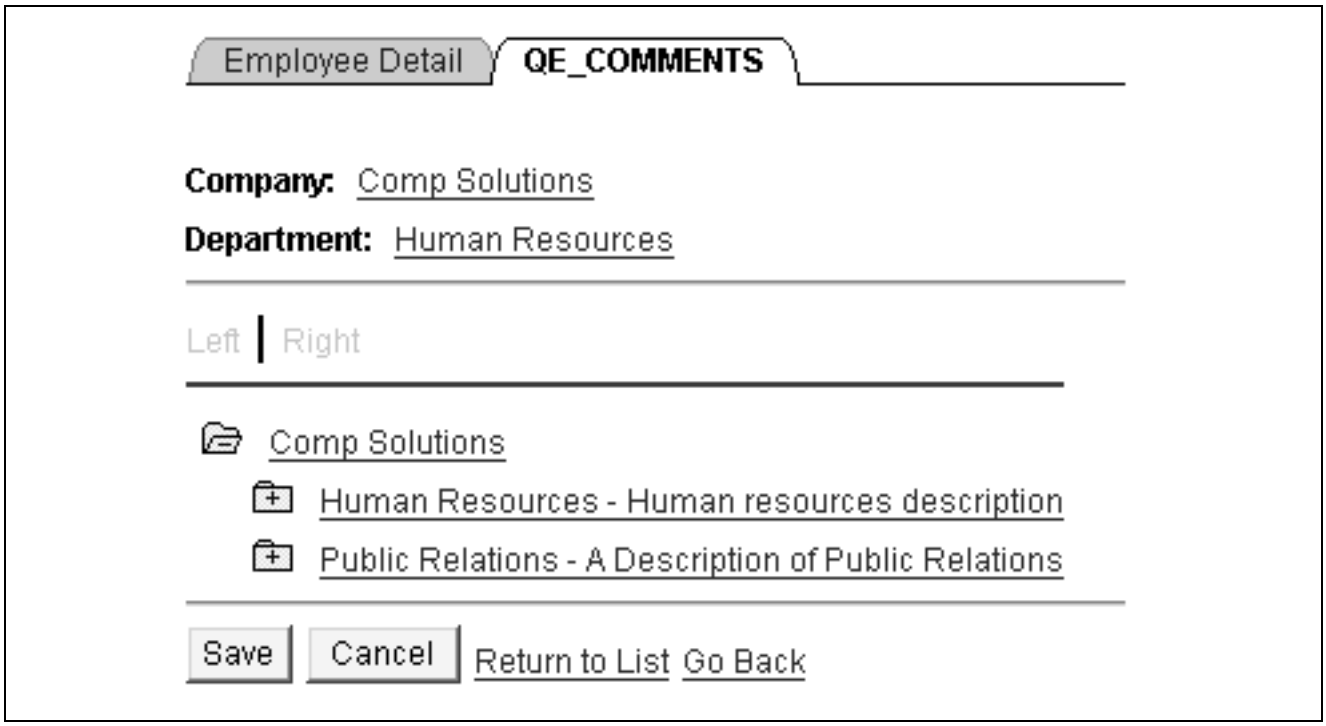
Syntax

```
GenerateMobileTree(&CIObject [,CIObject_property])
```

Description

Use the **GenerateMobileTree** function to display data in a tree format, with nodes and leaves. The result of the GenerateMobileTree function is an HTML string, which can be displayed in an HTML area control.

Use this function in the OnInit event to display the mobile tree, and on the OnChange event for programs related to expanding, collapsing or selecting nodes.



Mobile HTML Tree

The positional links at the top of the page (First, Previous, Next, Last, Left, Right) enable the user to navigate around the tree. These links are automatically generated as part of the execution of GenerateTree.

The icon next to a node can have a + sign or a - sign in it, depending on whether the node is collapsed or expanded. When a node is collapsed, none of the nodes that report to the collapsed node are displayed, and the icon has a + sign. When a node is expanded, all the nodes that report to it are displayed, and the icon has a - sign. You can collapse and expand a node by clicking in the icon. This enables you to view the tree at different levels of detail. When the icon for a node has no + or − sign in it, it is a terminal node, and cannot be expanded or collapsed.

Using the GenerateMobileTree Function

The **GenerateMobileTree** function is similar to the GenerateTree function, however, there are several important differences.

- The GenerateMobileTree function does not take a standalone rowset for data. Mobile does not support standalone rowsets, so you must populate your tree data one item at a time.
- You cannot use the Tree Classes with the GenerateMobileTree function.

To use the GenerateMobileTree function:

1. Define the Component Interface.

   You must use a Component Interface that is created from a component that contains the TREECTL_HDR and TREECTL_NODE records. When the Component Interface is synched to the device, these records do not have to contain any data, they can be populated using PeopleCode.

   See *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, The HTML Tree Rowset Records.

2. Add an HTML area to the mobile page.

This involves adding a long text field to the Component Interface, specifying how it displays on the mobile page, and so on. The value of the Component Interface long text field is what is displayed in the HTML area.

See Chapter 18, Mobile Classes , HTML Areas Used in Mobile, page 1016.

3. Determine what property on the Component Interface is going to contain the OnChange PeopleCode used to evaluate the end user action and call the appropriate PeopleCode programs.

   The action an end user takes (selecting a row, or expanding or collapsing a node) is accessible to this property.

   This property must be a character field that is at least 46 characters long. The field can be made invisible.

4. Specify this property as the value for the PAGE_FIELD_NAME field on the TREECTRL_HDR_SBR record.

5. Populate the tree, specify values for the nodes, leaves, images, and so on.

Parameters

Parameter	Description
&CIObject	This is the mobile object that is to be viewed in a hierarchy tree view. This object is from a peer reference.
CIObject_property	This is a property on the mobile object that is to be viewed in a hierarchy tree view.

Returns

None.

Example

In the OnInit event, generate the mobile tree.

```
Local ApiObject &ob;

&ob = %ThisMobileObject;

&MyCI = %Session.GetCompIntfc(ComIntfc.RDM_RELTREE);
&MyCI.Key_01 = "2";
&MyCI.Create();

&MyCI.PAGE_FIELD_NAME = "TREELEVELNT";

&MyCI.RDM.RELTREE_NDE.InsertItem(1);
&MyCI.RDM.RELTREE_NDE.Item(1).TREE_NODE = "NAME 1";
&MyCI.RDM.RELTREE_NDE.Item(1).DESCR = "Descr 1";
&MyCI.RDM.RELTREE_NDE.Item(1).PARENT_FLAG = "Y";
&MyCI.RDM.RELTREE_NDE.Item(1).TREE_LEVEL_NUM = 1;
&MyCI.RDM.RELTREE_NDE.Item(1).LEAF_FLAG = "N";

&MyCI.RDM.RELTREE_NDE.InsertItem(2);
&MyCI.RDM.RELTREE_NDE.Item(2).TREE_NODE = "NAME 3";
&MyCI.RDM.RELTREE_NDE.Item(2).DESCR = "Descr 2";
&MyCI.RDM.RELTREE_NDE.Item(2).PARENT_FLAG = "Y";
```

```
&MyCI.RDM.RELTREE_NDE.Item(2).TREE_LEVEL_NUM = 2;
&MyCI.RDM.RELTREE_NDE.Item(2).LEAF_FLAG = "N";

&MyCI.RDM.RELTREE_NDE.InsertItem(3);
&MyCI.RDM.RELTREE_NDE.Item(3).TREE_NODE = "NAME 1";
&MyCI.RDM.RELTREE_NDE.Item(3).DESCR = "Descr 3";
&MyCI.RDM.RELTREE_NDE.Item(3).PARENT_FLAG = "Y";
&MyCI.RDM.RELTREE_NDE.Item(3).TREE_LEVEL_NUM = 1;
&MyCI.RDM.RELTREE_NDE.Item(3).LEAF_FLAG = "N";

&OB.QE_COMMENTS = GenerateMobileTree(&MyCI);
```

The following PeopleCode is used in the OnChange event (used for when a user selects a node, expands a node or closes a node):

```
Local ApiObject &ob;

&ob = %ThisMobileObject;

&MyCI = %Session.GetCompIntfc(ComIntfc.RDM_RELTREE);
&MyCI.Key_01 = "2";
&MyCI.Get();

&MyCI.PAGE_FIELD_NAME = "TREECTLEVENT"

If Left(%ThisMobileObject.TREELEVENT, 1) = "S" Then
     Warning ("This node has been selected");
End-If;

&ob.QE_COMMENTS = GenerateMobileTree(&MyCI, %ThisMobileObject.TREELEVENT);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateTree , page 230

GenerateQueryContentURL

Syntax

GenerateQueryContentURL(**PORTAL.***portalname*, **NODE.***nodename*, *QueryName*, *IsPublic* [, *IsNewWindow*])

Description

The **GenerateQueryContentURL** function creates a URL string that represents an absolute reference the specified query (URL) on the content servlet.

The PortalURI of the node that hosts the specified portal is used in the generated URL. The generated URL contains a reference to the portal content (*psc*) servlet.

If you want to generate a relative URL, use **GenerateQueryRelativeURL**.

If you want to generate a URL for the portal service (*psp*) servlet, use **GenerateQueryPortalURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the query, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
Queryname	Specify the name of the query you want to generate a URL for. This parameter takes a string value.
IsPublic	Specify whether the query is public or private. This parameter takes a Boolean value: True, the query is public, False otherwise.
IsNewWindow	Specify whether the URL is for a new browser instance. This parameter takes a Boolean value: True, the URL is for a new browser instance, False otherwise. The default is False. If the value is True this function generates a new state block for use in a separate browser instance. This does *not* automatically open a new browser instance. It just supports it.

Note. When Query is being run on a PeopleTools version prior to 8.16, the query URL does *not* include the ability to specify if a query is public or private. On PeopleTools versions 8.16 and higher, the generated URL contains either the keyword PUBLIC or PRIVATE prepended to the query name. If you are building a URL for a portal node that is on a PeopleTools release prior to 8.16, you *must* remove the public or private keyword before trying to use the URL.

Returns

If *IsPublic* is specified as True, and the node has a Node Type of PIA, a string of the following format is returned:

```
http://PortalURI/Portal/node/q/?ICAction=ICQryNameURL=PUBLIC.QueryName
```

If *IsPublic* is specified as False, and the node has a Node Type of PIA, a string of the following format is returned:

```
http://PortalURI/Portal/node/q/?ICAction=ICQryNameURL=PRIVATE.QueryName
```

This function returns a Null string if you specify an invalid portal or node.

Example

The following code example:

```
&url = GenerateQueryContentURL(%Portal, "RMTNODE", "QUERYNAME", True);
```

might produce a string as follows:

```
http://bsto091200/psc/ps/EMPLOYEE/RMTNODE/q/?ICAction=ICQryNameURL=PUBLIC.QUERYNAME
```

The following code example uses the optional parameter to produce a URL that supports a new browser instance:

```
&url = GenerateQueryContentURL(%Portal, "RMTNODE", "QUERYNAME", True, True);
```

might produce a string as follows:

```
http://bsto091200/psc/ps_newwin/EMPLOYEE/RMTNODE/q/?ICAction=ICQryNameURL=⇒
```

```
PUBLIC.QUERYNAME
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateQueryPortalURL , page 220, Chapter 1, PeopleCode Built-in Functions, GenerateQueryRelativeURL , page 221, Chapter 15, Internet Script Classes (iScript) , page 931, Chapter 24, Query Classes , page 1303.
PeopleTools PeopleBook: PeopleSoft Query

GenerateQueryPortalURL

Syntax

GenerateQueryPortalURL(**PORTAL.***portalname*, **NODE.***nodename*, *QueryName*, *IsPublic* [, *IsNewWindow*])

Description

The **GenerateQueryPortalURL** function creates a URL string that represents an absolute reference the specified query (URL) on the portal servlet.

The PortalURI of the node that hosts the specified portal is used in the generated URL. The generated URL contains a reference to the portal service (*psp*) servlet.

If you want to generate a relative URL, use **GenerateQueryRelativeURL**.

If you want to generate a URL for the portal content (*psc*) servlet, use **GenerateQueryContentURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the query, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
Queryname	Specify the name of the query you want to generate a URL for. This parameter takes a string value.
IsPublic	Specify whether the query is public or private. This parameter takes a Boolean value: True, the query is public, False otherwise.
IsNewWindow	Specify whether the URL is for a new browser instance. This parameter takes a Boolean value: True, the URL is for a new browser instance, False otherwise. The default is False. If the value is True this function generates a new state block for use in a separate browser instance. This does *not* automatically open a new browser instance. It just supports it.

Note. When Query is being run on a PeopleTools version prior to 8.16, the query URL does *not* include the ability to specify if a query is public or private. On PeopleTools versions 8.16 and higher, the generated URL contains either the keyword PUBLIC or PRIVATE prepended to the query name. If you are building a URL for a portal node that is on a PeopleTools release prior to 8.16, you *must* remove the public or private keyword before trying to use the URL.

Returns

If *IsPublic* is specified as True, and the node has a Node Type of PIA, a string of the following format is returned:

```
http://PortalURI/Portal/node/q/?ICAction=ICQryNameURL=PUBLIC.QueryName
```

If *IsPublic* is specified as False, and the node has a Node Type of PIA, a string of the following format is returned:

```
http://PortalURI/Portal/node/q/?ICAction=ICQryNameURL=PRIVATE.QueryName
```

This function returns a Null string if you specify an invalid portal or node.

Example

The following code example:

```
&url = GenerateQueryPortalURL(%Portal, "RMTNODE", "QUERYNAME", True);
```

might produce a string as follows:

```
http://bsto091200/psp/ps/EMPLOYEE/RMTNODE/q/?ICAction=ICQryNameURL=PUBLIC.QUERYNAME
```

The following code example uses the optional parameter to produce a URL that supports a new browser instance:

```
&url = GenerateQueryPortalURL(%Portal, "RMTNODE", "QUERYNAME", True, True);
```

might produce a string as follows:

```
http://bsto091200/psp/ps_newwin/EMPLOYEE/RMTNODE/q/?ICAction=ICQryNameURL=⇒
PUBLIC.QUERYNAME
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateQueryContentURL , page 218, Chapter 1, PeopleCode Built-in Functions, GenerateQueryRelativeURL , page 221, Chapter 15, Internet Script Classes (iScript) , page 931, Chapter 24, Query Classes , page 1303.
PeopleTools PeopleBook: PeopleSoft Query

GenerateQueryRelativeURL

Syntax

GenerateQueryRelativeURL(**PORTAL.***portalname*, **NODE.***nodename*, *QueryName*, *IsPublic* [, *IsNewWindow*])

Description

The **GenerateQueryRelativeURL** function creates a URL string that represents a relative reference to the specified query on the portal servlet. The relative reference is suitable for use on any page that itself has the simple URL format.

If you want to generate an absolute URL, use either **GenerateQueryPortalURL** or **GenerateQueryContentURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the query, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
Queryname	Specify the name of the query you want to generate a URL for. This parameter takes a string value.
IsPublic	Specify whether the query is public or private. This parameter takes a Boolean value: True, the query is public, False otherwise.
IsNewWindow	Specify whether the URL is for a new browser instance. This parameter takes a Boolean value: True, the URL is for a new browser instance, False otherwise. The default is False. If the value is True this function generates a new state block for use in a separate browser instance. This does *not* automatically open a new browser instance. It just supports it.

Note. When Query is being run on a PeopleTools version prior to 8.16, the query URL does *not* include the ability to specify if a query is public or private. On PeopleTools versions 8.16 and higher, the generated URL contains either the keyword PUBLIC or PRIVATE prepended to the query name. If you are building a URL for a portal node that is on a PeopleTools release prior to 8.16, you *must* remove the public or private keyword before trying to use the URL.

Returns

If *IsPublic* is specified as True, and the node has a Node Type of PIA, a string of the following format is returned:

```
../../../portal/node/q/?ICAction=ICQryNameURL=PUBLIC.QueryName
```

If *IsPublic* is specified as False, and the node has a Node Type of PIA, a string of the following format is returned:

```
../../../portal/node/q/q/?ICAction=ICQryNameURL=PRIVATE.QueryName
```

This function returns a Null string if you specify an invalid portal or node.

Example

The following code example:

```
&url = GenerateQueryRelativeURL(%Portal, "RMTNODE", "QUERYNAME", True);
```

might produce a string as follows:

```
../../../EMPLOYEE/RMTNODE/q/?ICAction=ICQryNameURL=PUBLIC.QUERYNAME
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateQueryContentURL , page 218, Chapter 1, PeopleCode Built-in Functions, GenerateQueryPortalURL , page 220, Chapter 15, Internet Script Classes (iScript) , page 931, Chapter 24, Query Classes , page 1303.
PeopleTools PeopleBook: PeopleSoft Query

GenerateScriptContentRelURL

Syntax

```
GenerateScriptContentRelURL(PORTAL.portalname, NODE.nodename, RECORD.recordname,
FIELD.fieldname, event_name, function_name, [, keylist])
```

where *keylist* is a list of field references in the form:
`[recordname.]field1 [, [recordname.]field2]. . .`

OR
`&RecordObject1 [, &RecordObject2]. . .`

Description

The **GenerateScriptContentRelURL** function creates a URL string that represents a relative reference to the specified iScript. The generated URL contains a reference to the portal content (*psc*) servlet.

If you want to generate an absolute URL for an iScript for the portal content servlet, use **GenerateScriptContentURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the iScript, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
recordname	Specify the name of the record containing the iScript, prefixed with the reserved word **RECORD**.
fieldname	Specify the name of the field containing the iScript, prefixed with the reserved word **FIELD**.
event_name	Specify the name of the event containing the iScript. This is generally the FieldFormula event.
function_name	Specify the name of the iScript function.
keylist	An optional list of parameters used with the function. It can also be an already instantiated record object.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

```
/psc/s/recname.fieldname.event_name.function_name?parameters
```

If the node has a Node Type of ICType, a string of the following format is returned:

```
/portal/node/?ICType=Script&ICScriptProgramName=recname.fieldname.event_⇒
name.function_name?parameters
```

The question mark and the text following the question mark may or may not be included, depending on whether or not you specified a page and action or not.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&MyScriptURL = GenerateScriptContentRelURL("EMPLOYEEPORTAL", "CRM", ⇒
Record.WEBLIB_CRM, Field.SFASCRIPTS, "FieldFormula", "Iscript_SFAHOME ");
```

Might yield the following URL:

```
/psc/s/WEBLIB_CRM.SFASCRIPTS.FieldFormula.IScript_SFAHOME
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentURL , page 224, Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalRelURL , page 226, Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalURL , page 227, Chapter 1, PeopleCode Built-in Functions, GenerateScriptRelativeURL , page 229, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateScriptContentURL

Syntax

GenerateScriptContentURL(**PORTAL**.*portalname*, **NODE**.*nodename*, **RECORD**.*recordname*, **FIELD**.*fieldname*, *event_name*, *function_name*, [, *keylist*])

where *keylist* is a list of field references in the form:
[*recordname*.]*field1* [, [*recordname*.]*field2*]. . .

OR
&RecordObject1 [, *&RecordObject2*]. . .

Description

The **GenerateScriptContentURL** function creates a URL string that represents an absolute reference to the specified iScript for the content servlet.

The ContentURI of the specified node is used in the generated URL. The URL contains a reference to the portal content (*psc*) servlet.

If you want to generate a URL for an iScript for the portal servlet, use GenerateScriptPortalURL.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the iScript, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
recordname	Specify the name of the record containing the iScript, prefixed with the reserved word **RECORD**.
fieldname	Specify the name of the field containing the iScript, prefixed with the reserved word **FIELD**.
event_name	Specify the name of the event containing the iScript. This is generally the FieldFormula event.
function_name	Specify the name of the iScript function.
keylist	An optional list of parameters used with the function. It can also be an already instantiated record object.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

```
http://Content URI of host node/portal/node/s/recname.fieldname.event_⇒
name.function_name?parameters
```

If the node has a Node Type of ICType, a string of the following format is returned:

```
http://Content URI of host node/portal/node/?ICType=Script&ICScriptProgramName=⇒
recname.fieldname.event_name.function_name?parameters
```

The question mark and the text following the question mark may or may not be included, depending on whether or not you specified a page and action or not.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&MyScriptURL = GenerateScriptContentURL("EMPLOYEEPORTAL", "CRM", ⇒
Record.WEBLIB_CRM, Field.SFASCRIPTS, "FieldFormula", "Iscript_SFAHOME ");
```

Might yield the following URL:

```
http://mike.com/servlets/psc/testsite/EMPLOYEEPORTAL/CRM/s/WEBLIB_⇒
CRM.SFASCRIPTS.FieldFormula.IScript_SFAHOME
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentRelURL , page 223, Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalRelURL , page 226, Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalURL , page 227, Chapter 1, PeopleCode Built-in Functions, GenerateScriptRelativeURL , page 229, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateScriptPortalRelURL

Syntax

GenerateScriptPortalRelURL(**PORTAL**.*portalname*, **NODE**.*nodename*, **RECORD**.*recordname*, **FIELD**.*fieldname*, *event_name*, *function_name*, [, *keylist*])

where *keylist* is a list of field references in the form:
[*recordname*.]*field1* [, [*recordname*.]*field2*]. . .

OR
&RecordObject1 [, *&RecordObject2*]. . .

Description

The **GenerateScriptPortalRelURL** function creates a URL string that represents a relative reference to the specified iScript. The generated URL contains a reference to the portal service (*psp*) servlet.

If you want to generate an absolute URL for an iScript for the portal service servlet, use **GenerateScriptPortalURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the iScript, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
recordname	Specify the name of the record containing the iScript, prefixed with the reserved word **RECORD**.
fieldname	Specify the name of the field containing the iScript, prefixed with the reserved word **FIELD**.
event_name	Specify the name of the event containing the iScript. This is generally the FieldFormula event.
function_name	Specify the name of the iScript function.
keylist	An optional list of parameters used with the function. It can also be an already instantiated record object.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

*/psp/***s**/*recname.fieldname.event_name.function_name***?***parameters*

If the node has a Node Type of ICType, a string of the following format is returned:

*/portal/node/***?ICType=Script&ICScriptProgramName=***recname.fieldname.event_*⇒
*name.function_name***?***parameters*

The question mark and the text following the question mark may or may not be included, depending on whether or not you specified a page and action or not.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&MyScriptURL = GenerateScriptPortalRelURL("EMPLOYEEPORTAL", "CRM", ⇒
Record.WEBLIB_CRM, Field.SFASCRIPTS, "FieldFormula", "IScript_SFAHOME");
```

Might yield the following:

```
/psp/s/WEBLIB_CRM.SFASCRIPTS.FieldFormula.IScript_SFAHOME
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentRelURL , page 223, Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentURL , page 224, Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalURL , page 227, Chapter 1, PeopleCode Built-in Functions, GenerateScriptRelativeURL , page 229, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateScriptPortalURL

Syntax

GenerateScriptPortalURL(**PORTAL**.*portalname*, **NODE**.*nodename*, **RECORD**.*recordname*, **FIELD**.*fieldname*, *event_name*, *function_name*, [, *keylist*])

where *keylist* is a list of field references in the form:
[*recordname*.]*field1* [, [*recordname*.]*field2*]. . .

OR
&RecordObject1 [, *&RecordObject2*]. . .

Description

The **GenerateScriptPortalURL** function creates a URL string that represents an absolute reference to the specified iScript for the portal servlet. The PortalURI of the node that hosts the specified portal is used in the generated URL. The URL contains a reference to the portal service (*psp*) servlet.

If you want to generate a URL for an iScript for the portal content (*psc*) servlet, use **GenerateScriptContentURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the iScript, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
recordname	Specify the name of the record containing the iScript, prefixed with the reserved word **RECORD**.
fieldname	Specify the name of the field containing the iScript, prefixed with the reserved word **FIELD**.
event_name	Specify the name of the event containing the iScript. This is generally the FieldFormula event.
function_name	Specify the name of the iScript function.
keylist	An optional list of parameters used with the function. It can also be an already instantiated record object.

Returns

If a node has a Node Type of PIA, a string of the following format is returned:

```
http://Portal URI of host portal/portal/node/s/recname.fieldname.event_⇒
name.function_name?parameters
```

If the node has a Node Type of ICType, a string of the following format is returned:

```
http://Portal URI of host node/portal/node/?ICType=Script&ICScriptProgramName=⇒
recname.fieldname.event_name.function_name?parameters
```

The question mark and the text following the question mark may or may not be included, depending on whether or not you specified a page and action or not.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&MyScriptURL = GenerateScriptPortalURL("EMPLOYEEPORTAL", "CRM", Record.WEBLIB_CRM,⇒
 Field.SFASCRIPTS, "FieldFormula", "IScript_SFAHOME");
```

Might yield the following:

```
http://mike.com/servlets/psp/testsite/EMPLOYEEPORTAL/CRM/s/WEBLIB_⇒
CRM.SFASCRIPTS.FieldFormula.IScript_SFAHOME
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentRelURL , page 223, Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentURL , page 224, Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalRelURL , page 226, Chapter 1, PeopleCode Built-in Functions, GenerateScriptRelativeURL , page 229, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateScriptRelativeURL

Syntax

GenerateScriptRelativeURL(**PORTAL**.*portalname*, **NODE**.*nodename*, **RECORD**.*recordname*, **FIELD**.*fieldname*, *event_name*, *function_name*, [, *keylist*])

where *keylist* is a list of field references in the form:
[*recordname*.]*field1* [, [*recordname*.]*field2*]. . .

OR
&RecordObject1 [, *&RecordObject2*]. . .

Description

The **GenerateScriptRelativeURL** function creates a relative URL string that represents a relative reference to the specified iScript. The relative reference is suitable for use on any page that has the simple URL format.

If you want to generate an absolute URL for an iScript, use either **GenerateScriptContentURL** or **GenerateScriptPortalURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the iScript, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
recordname	Specify the name of the record containing the iScript, prefixed with the reserved word **RECORD**.
fieldname	Specify the name of the field containing the iScript, prefixed with the reserved word **FIELD**.
event_name	Specify the name of the event containing the iScript. This is generally the FieldFormula event.
function_name	Specify the name of the iScript function.
keylist	An optional list of parameters used with the function. It can also be an already instantiated record object.

Returns

If the node has a Node Type of PIA, a string of the following format is returned:

*portal/node/***s**/*recname.fieldname.event_name.function_name***?***parameters*

If the node has a Node Type of ICType, a string of the following format is returned:

*portal/node/***?ICType=Script&ICScriptProgramName=***recname.fieldname.event_*⇒
*name.function_name***?***parameters*

The question mark and the text following the question mark may or may not be included, depending on whether or not you specified a page and action or not.

This function returns a Null string if you specify an invalid portal or node.

Example

The following code:

```
&MyScriptURL = GenerateScriptRelativeURL("EMPLOYEEPORTAL", "CRM", ⇒
Record.WEBLIB_CRM, Field.SFASCRIPTS, "FieldFormula", "IScript_SFAHOME");
```

Might yield the following:

```
../../../EMPLOYEEPORTAL/CRM//s/WEBLIB_CRM.SFASCRIPTS.FieldFormula.IScript_SFAHOME
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentRelURL , page 223, Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentURL , page 224, Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalRelURL , page 226, Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalURL , page 227, Chapter 15, Internet Script Classes (iScript) , page 931.

GenerateTree

Syntax

GenerateTree(*&rowset* [, *TreeEventField*])

Description

The GenerateTree function is used to display data in a tree format, with nodes and leaves. The result of the GenerateTree function is an HTML string, which can be displayed in an HTML area control. The tree generated by GenerateTree is called an *HTML tree*.

The GenerateTree function can be used in conjunction with the Tree Classes to display data from trees created using Tree Manager.

The GenerateTree function works with both an HTML area control and a hidden field. The *TreeEventField* parameter contains the contents of the invisible character field used to process the HTML tree events.

When an end-user selects a node, expands a node, collapses a node, or uses one of the navigation links, that event (end-user action) is passed to the invisible field, and the invisible field's FieldChange PeopleCode is executed.

See Also

PeopleCode Developer's Guide, Using Methods and Built-In Functions, Using HTML Tree End-User Actions (Events), *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Building Your HTML Tree Page.

Parameters

Parameter	Description
&rowset	Specify the name of the rowset you've populated with tree data.
TreeEventField	Specify the contents of the invisible character field used to process the HTML tree events. The first time the GenerateTree function is used, that is, to generate the initial tree, you don't need to include this parameter. Subsequent calls require this parameter.

Returns

A string that contains HTML code that can be used with the HTML control to display a tree.

Example

In the following example, TREECTLEVENT is the name of the invisible control field that contains the event string that was passed from the browser.

```
HTMLAREA = GenerateTree(&TREECTL, TREECTLEVENT);
```

See Also

PeopleCode Developer's Guide, Using Methods and Built-In Functions, Using the GenerateTree Function,
PeopleCode Developer's Guide, Using Methods and Built-In Functions, PostBuild PeopleCode Example,
PeopleCode Developer's Guide, Using Methods and Built-In Functions, FieldChange PeopleCode Example.

GenerateWorklistPortalURL

Syntax

GenerateWorklistPortalURL(**PORTAL**.*portalname*, **NODE**.*nodename*, *BusProc*, *Activity*, *Event*, *Worklist*, *Instance*)

Description

The **GenerateWorklistPortalURL** function creates a URL string that represents an absolute reference the specified Worklist (URL) on the portal servlet.

The PortalURI of the node that hosts the specified portal is used in the generated URL. The generated URL contains a reference to the portal service (*psp*) servlet.

If you want to generate a relative URL, use **GenerateWorklistRelativeURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
BusProc	Specify the business process of the Worklist.
Activity	Specify the activity of the Worklist.
Event	Specify the event of the Worklist.
Instance	Specify the instance of the Worklist.

Returns

A string of the following format:

```
http://PortalURI/Portal/node/w/BusProc.Activity.Event.Worklist.Instance
```

This function returns a Null string if you specify an invalid portal or node.

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateWorklistRelativeURL , page 232, Chapter 15, Internet Script Classes (iScript) , page 931, *PeopleSoft Workflow*, Introducing PeopleSoft Workflow.

GenerateWorklistRelativeURL

Syntax

GenerateWorklistRelativeURL(**PORTAL**.*portalname*, **NODE**.*nodename*, *BusProc*, *Activity*, *Event*, *Worklist*, *Instance*)

Description

The **GenerateWorklistRelativeURL** function creates a URL string that represents a relative reference to the specified Worklist on the portal servlet. The relative reference is suitable for use on any page that itself has the simple URL format.

If you want to generate an absolute URL, use **GenerateWorklistPortalURL**.

Parameters

Parameter	Description
portalname	Specify the name of the portal used for this request, prefixed with the reserved word **PORTAL**. You can also use a string, such as %Portal, for this value.
nodename	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
BusProc	Specify the business process of the Worklist.
Activity	Specify the activity of the Worklist.
Event	Specify the event of the Worklist.
Instance	Specify the instance of the Worklist.

Returns

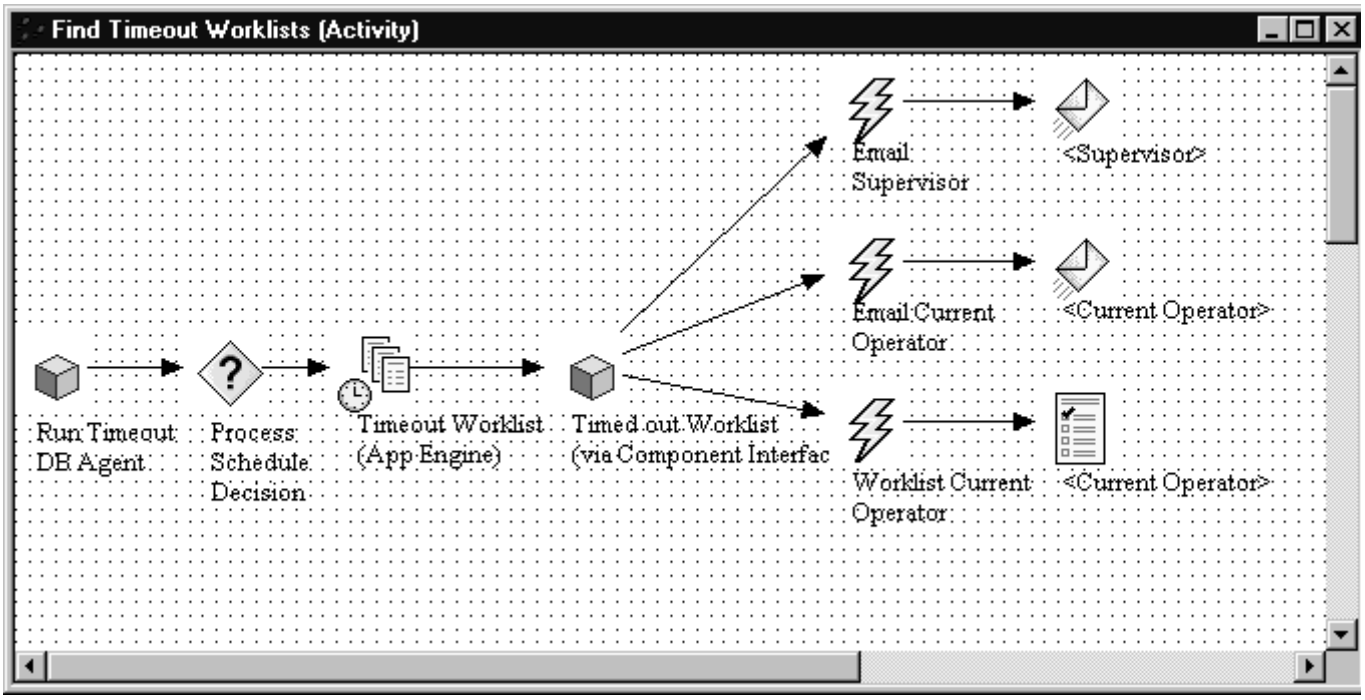
A string of the following format:

```
../../../Portal/Node/w/BusProc.Activity.Event.Worklist.Instance
```

This function returns a Null string if you specify an invalid portal or node.

Example

Given the following activity:



Example activity

The following is an example PeopleCode statement used to access this activity:

```
GenerateWorklistRelativeURL(%Portal, %Node, "Administer Workflow", "Find Timeout⇒
 Worklists", "Worklist Current Operator", "Timeout Notification", 1);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateWorklistPortalURL , page 231, Chapter 15, Internet Script Classes (iScript) , page 931, *PeopleSoft Workflow*, Introducing PeopleSoft Workflow.

GetAESection

Syntax

GetAESection(*ae_applid*, *ae_section* [, *effdt*])

Description

Opens and associates an AESection PeopleCode object with the **base** section, as specified. If no base section by the specified name is found, one is created. This enables you to create base sections as needed.

Warning! When you open or get an AESection object, (that is, the base section) any existing steps in the section are deleted.

After you've instantiated the AESection object, set the **template** section using the **SetTemplate** method. You can copy steps *from* the template section *to* the base section before you start the PeopleSoft Application Engine program. This is useful for applications that let users input their "rules" into a user-friendly page, then convert these rules, at save time, into Application Engine constructs.

When an AESection is opened (or accessed), the system first looks to see if it exists with the given input parameters. If such a section doesn't exist, the system looks for a similar section based on market, database platform, and effective date.

The AESection Object is designed for use within online. Typically, dynamic sections should be constructed in response to an end user action.

Note. Do not call an AESection object from an Application Engine PeopleCode Action. If you need to access another section, use the CallSection action instead.

Parameters

Parameter	Description
ae_applid	Specifies the application ID of the section you want to modify.
ae_section	Specifies the section name of the base section you want to modify. If no base section by the specified name is found, one is created.
effdt	Specifies the effective date of the section you want to modify (optional).

Returns

An AESection object is returned.

See Also

Chapter 5, AESection Class , page 591, Chapter 5, AESection Class , Open, page 597, Chapter 5, AESection Class , AESection Example , page 593.

GetAttachment

Syntax

```
GetAttachment(URLSource, SysFileName, LocalFile [, LocalDirEnvVar])
```

where *URLSource* can have **one** of the following forms:

```
URL.URLname
```

OR a string URL, such as

```
ftp://user:password@ftp.ps.com/
```

Description

The **GetAttachment** function transfers a file from the file server to the application sever.

File Name Considerations

When the file is transferred, the following characters are replaced with an underscore:

- space
- semicolon
- plus sign
- percent sign

- ampersand
- apostrophe
- exclamation point
- pound sign
- dollar sign

Note. The @ sign is *not* changed to an underscore on files transferred with this function.

Parameters

Parameter	Description
URLSource	A reference to a URL. This can be either a URL name, in the form URL.*URLName*, or a string. This is where the file is transferred to.
SysFileName	The relative path and filename of the file on the file server. This is appended to *URLSource* to make up the full URL where the file is transferred from. This parameter takes a string value. **Note.** The *URLSource* requires "/" slashes. Because *SysFileName* is appended to the URL, it also requires only "/" slashes. "\" are NOT supported in anyway for either the *URLSource* or the *SysFileName* parameter.
LocalFile	The name, and possible full path, to the destination file on the application server. This parameter takes a string value. If you specify only a file name, the file is assumed to be in the directory specified by the **TMP** environment variable or the directory specified by *LocalDirEnvVar*. If the file is *not* in the **TMP** directory and you don't specify *LocalDirEnvVar* you must use the full path name as appropriate to the operating system (that is, on Unix use "\").
LocalDirEnvVar	The name of the environment variable containing path information. If specified, the path value in the environment variable is prepended to *LocalFile*. Use this if you don't want to hard code path names in the *LocalFile* parameter. This parameter takes a string value. **Note.** Note. Only operating system environment variables are recognized with this parameter, rather than variables defined in configuration files.

Returns

An integer value. You can check for either the integer or the constant:

Numeric Value	Constant Value	Description
0	%Attachment_Success	File was transferred successfully.
1	%Attachment_Failed	File was not successfully transferred.

Example

The following transfers the file "resume.txt" from the file server to the application server using ftp.

```
GetAttachment("ftp://anonymous:hobbit1@ftp.ps.com/HRarchive/", ⇒
"NewHire/11042000resume.txt", "c:/NewHires/resume.txt";
```

In this example, all resumes are in the directory NewHires, which is pointed to by the environment variable RESUMES.

```
&SysFileName = %DATE  | &LASTNAME;

GetAttachment(URL.MYFTP, &SysFileName, "resume.txt", "RESUMES");
```

The following example gets employee photos from an ftp archive, and writes them to a record.

```
Local File &FILE;
Local Record &REC;
Local SQL &SQL;

&REC = CreateRecord(Record.EMPL_PHOTO);
&SQL = CreateSQL("%SelectAll(:1)", Record.EMPL_PHOTO);

While &SQL1.Fetch(&REC)

/* photos are stored in FTP archive by EmplID */
   &EmplId = &REC.EmplId.Value;
   &SysFileName = &EmplId | "_PHOTO.GIF";

/* Store files with same name on local system */
   GetAttachment(URL.FTPARCHIVE, &SysFileName, &SysFileName, "PHOTOS");

/* get the file */
   &FileName = "C:\HRArchive\Photos" | &SysFileName;
   &FILE = GetFile(&FileName, "w", "a", %FilePath_Absolute);

/* Write file to record */
   &FILE.WriteRaw(&REC.EMPLOYEE_PHOTO.Value);

End-While;

&FILE.Close();
```

See Also

Chapter 1, PeopleCode Built-in Functions, AddAttachment , page 31, Chapter 1, PeopleCode Built-in Functions, DeleteAttachment , page 135, Chapter 1, PeopleCode Built-in Functions, PutAttachment , page 353, Chapter 1, PeopleCode Built-in Functions, ViewAttachment , page 494, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Understanding File Attachment Architecture.

GetBiDoc

Syntax

GetBiDoc(*XMLstring*)

Description

The **GetBiDoc** function creates a BiDocs structure. You can populate the structure with data from *XMLstring*. This is part of the incoming Business Interlink functionality, which enables PeopleCode to receive an XML request and return an XML response.

Parameters

Parameter	Description
XMLstring	A string containing XML. You can specify a NULL string for this parameter, that is, two quotation marks ("") without a space between them.

Return Value

A BiDocs object.

Example

The following example gets an XML request, puts it into a text string, and puts that into a BiDoc. After this is done, the GetDoc method and the GetValue method can get the value of the skills XML element, which is contained within the postreq XML element in the XML request.

```
Local BIDocs &rootInDoc, &postreqDoc;
Local string &blob;
Local number &ret;

&blob = %Request.GetContentBody();
/* process the incoming xml(request)- Create a BiDoc and fill with the request*/
&rootInDoc = GetBiDoc(&blob);
&postreqDoc = &rootInDoc.GetDoc("postreq");
&ret = &postreqDoc.GetValue("skills", &skills);
```

You can also create an empty BiDoc with GetBiDoc, as in the following example.

```
Local BIDocs &rootInDoc, &postreqDoc;
Local string &blob;
Local number &ret;

&blob = %Request.GetContentBody();
/* process the incoming xml(request)- Create a BiDoc and fill with the request*/
&rootInDoc = GetBiDoc("");
&ret = &rootInDoc.ParseXMLString(&blob);
&postreqDoc = &rootInDoc.GetDoc("postreq");
&ret = &postreqDoc.GetValue("skills", &skills);
```

See Also

PeopleTools PeopleBook: PeopleSoft Business Interlink Application Developer's Guide

GetCalendarDate

Syntax

GetCalendarDate(*comparedate*, *periods*, *periodadjustment*, *outfieldname*, *company*, *paygroup*)

Description

GetCalendarDate returns the value of a Date field from the PS_PAY_CALENDAR table. If a table entry is not found, GetCalendarDate returns 1899-01-01.

Processing Rules

The following are the processing rules for GetCalendarDate:

1. The function SELECTs all the values for *outfieldname*, PAY_BEGIN_DT and PAY_END_DT from PS_PAY_CALENDAR. The result set is sorted in increasing PAY_END_DT order.
2. A SQL SELECT statement is generated in the following form:
3. SELECT outfieldname, PAY_BEGIN_DT, PAY_END_DT FROM PS_PAY_CALENDAR WHERE COMPAny=:1 AND PAYGROUP=:2 ORDER BY PAY_END_DT;
4. Rows are fetched from the result set until the value of *comparedate* falls between PAY_BEGIN_DT and PAY_END_DT. The value of *outfieldname* is stored in a storage stack.
5. A work variable equal to the value in *periods* is set.
6. If the value of *outfieldname* in the located result row is equal to *comparedate*, then the value in *periodadjustment* is added to the work variable. Because *periodadjustment* may be negative, the result may be negative.
7. If the work variable is negative then the saved value of *outfieldname* is returned from the storage stack at the level specified by the work variable. If the work variable is positive then fetch forward the number of times specified by the work variable. The value of *outfieldname* is returned from the most recently fetched (current) row.

Parameters

Parameter	Description
comparedate	A date field set by the caller as the date of interest, for example, "1997-02-17."
periods	A numeric variable set by the caller specifying the number of periods forward or backward to be returned.
periodadjustment	A numeric variable that adjusts the *periods* if the *comparedate* equals the period end date. This is typically used to adjust for period end dates. Usually the *periodadjustment* is either -1, 0, or 1.
outfieldname	The name of a date field in the PS_PAY_CALENDAR table. For example PAY_BEGIN_DT. The value of this field is not referenced or modified by the routine, but the name of the field is used to build a SQL SELECT statement and to indicate which value from the table to return in the return date.
company	A field set by the caller to be equal to the company code of interest, for example, "CCB".
paygroup	A variable set by the caller to be equal to the PayGroup code of interest, for example, "M01".

Returns

Returns a Date value from the PS_PAY_CALENDAR table.

Example

The following examples use the sample PS_PAY_CALENDAR entries in the following table. In the example, *comparedate* and the result date are Date type fields defined in some record.

COMPANY	PAYGROUP	PAY_END_DT	PAY_BEGIN_DT	CHECK_DT
CCB	MO1	1997-01-31	1997-01-01	1997-01-31
CCB	MO1	1997-02-28	1997-02-01	1997-02-28
CCB	MO1	1997-03-31	1997-03-01	1997-03-29
CCB	MO1	1997-04-30	1997-04-01	1997-04-30
CCB	MO1	1997-05-31	1997-05-01	1997-05-31
CCB	MO1	1997-06-30	1997-06-01	1997-06-28
CCB	MO1	1997-07-31	1997-07-01	1997-07-31
CCB	MO1	1997-08-31	1997-08-01	1997-08-30
CCB	MO1	1997-09-30	1997-09-01	1997-09-30
CCB	MO1	1997-10-31	1997-10-01	1997-10-31
CCB	MO1	1997-11-30	1997-11-01	1997-11-27
CCB	MO1	1997-12-31	1997-12-01	1997-12-31
CCB	SM1	1997-01-15	1997-01-01	1997-01-15

Find the begin date of the pay period containing the date 1997-05-11 (the value of &COMPAREDate). The result date returned would be 1997-05-01.

```
&RESULT_Date = GetCalendarDate(&COMPAREDate, 0, 0,
   PAY_BEGIN_DT, COMPAny, PAYGROUP);
```

Or:

```
&RESULT_Date = GetCalendarDate(&COMPAREDate, 1, -1,
   PAY_BEGIN_DT, COMPAny, PAYGROUP);
```

GetChart

Syntax

GetChart(*RecordName.FieldName*)

Description

Use the **GetChart** function to get a reference to a chart object. You associate a record and field name with a chart definition in Application Designer.

Parameters

Parameter	Description
RecordName.FieldName	Specify the record and field name associated with the chart.

Returns

A reference to a chart object.

Example

```
&MyChart = GetChart(Chart_Record.Chart_Field);
```

See Also

Chapter 9, Chart Class , page 725, Chapter 9, Chart Class , Creating a Chart , page 766.

GetChartURL

Syntax

GetChartURL(*&Chart*)

Description

Use the **GetChartURL** method to generate the URL of a chart object. This URL can then be used in your application for displaying a chart.

Parameters

Parameter	Description
&Chart	Specify an already instantiated chart object.

Returns

A URL as a string.

Example

```
Function IScript_GetChartURL()
local object &MyChart;
local string &MyURL;

   &MyChart = CreateObject("Chart");
   &MyChart .SetData = xx;

/* xx will be a data row set */
```

```
      &MyURL = %Response.GetChartURL(&MyChart);
      &sMap = &oChart.ImageMap;

      %Response.Write("<HTML><IMG SRC=");
      %Response.Write(&MyURL);
      %Response.Write("  USEMAP=#THEMAP></IMG><MAP NAME=THEMAP>");
      %Response.Write(&sMap);
      %Response.Write("</MAP></HTML>");

   End-Function;
```

See Also

Chapter 9, Chart Class , page 725.

GetCwd

Syntax

GetCwd()

Description

GetCwd determines the current working directory of the process that executes it. This means that in PIA it returns the current working directory on the server, in an Application Engine program it returns the current working directory of the Application Engine process, and so on.

Returns

Returns a string containing the path of the current working directory.

Example

The example stores a string specifying the current working directory in &CWD.

```
&CWD = GetCwd();
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetEnv , page 241, Chapter 1, PeopleCode Built-in Functions, ExpandEnvVar , page 178.

GetEnv

Syntax

GetEnv(*env_var_str*)

Description

This function returns the value of an environment variable specified by *env_var_str* as a string. If the environment variable does not exist, it returns a null string.

You can use the GetEnv function to determine the actual path of PS_HOME. You could use this with the Exec function, which automatically prepends the command string with the path of PS_HOME.

Parameters

Parameter	Description
env_var_str	A string containing an environment variable.

Returns

Returns a string equal to *env_var_str* with any enclosed environment variables expanded to their values.

Example

Assume that the environment string NETDRIVE is equal to "N:". The following statement returns "N:" in &NETDRIVE:

```
&NETDRIVE = GetEnv("netdrive");
```

Note that on UNIX systems, the environment variable names are case-sensitive. That is, "netdrive" is a different variable from "NetDrive".

See Also

Chapter 1, PeopleCode Built-in Functions, GetCwd , page 241, Chapter 1, PeopleCode Built-in Functions, ExpandEnvVar , page 178.

GetField

Syntax

GetField([*recname.fieldname*])

Description

GetField creates a reference to a field object for the current context; that is, from the row containing the currently executing program.

If you don't specify *recname.fieldname* the current field executing the PeopleCode is returned.

Note. For PeopleCode programs located in events that are not associated with a specific row, record, and field at the point of execution this function is invalid. That is, you cannot use this function in PeopleCode programs on events associated with high-level objects like pages or components. For events associated with record level programs (like component record), this function is valid, but it must be specified with a field name, as there is an assumed record, but no assumed field name.

When GetField is used with an associated record and field name on component buffer data, the following is assumed:

```
&FIELD = GetRow().recname.fieldname;
```

Parameters

Parameter	Description
recname.fieldname	If you don't want to refer to the field executing the PeopleCode, specify a record name and field name.

Returns

This function returns a field object that references the field from the specified record.

Example

```
Local Field &CHARACTER;

&CHARID = GetField(FIELD.CHAR_ID);
```

See Also

Chapter 12, Field Class , page 843, Chapter 25, Record Class , GetField, page 1450,
PeopleCode Developer's Guide, Data Buffer Access.

GetFile

Syntax

GetFile(*filename, mode [, charset*] [*, pathtype*])

Description

The **GetFile** function instantiates a new file object from the File class, associates it with an external file, and opens the file so you can use File class methods to read from or write to it.

GetFile is similar to the **Open** File Class method, but Open associates an *existing* file object with an external file. If you plan to access only one file at a time, you need only one file object. Use GetFile to instantiate a file object for the first external file you access, then use Open to associate the same file object with as many different external files as you want. You need to instantiate multiple file objects with GetFile only if you expect to have multiple files open at the same time.

Working With Relative Paths

If you specify a relative path, that path is appended to the path specified by an environmental variable, if the variable exists, in a particular order. This means if the first environmental variable in order is found, that is the one that's used. If it isn't found, the next one is searched for, and if found, is used, and so on.

The environmental variables that are used are different, depending on whether the machine where the code is running is an application server. If the code is running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]

If the code is *not* running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]

2. PS_SERVDIR\Files[\relative_path]
3. TEMP[\relative_path]

The PS_FILEDIR environmental variable is *not* initialized and set to a path automatically. If you want to use this environmental variable, you must set it yourself.

Note. Your system security should verify if a user has the correct permissions before allowing access to a drive (for example, if a user changes their **TEMP** environmental variable to a network drive they don't normally have access to, your system security should detect this.)

Parameters

Parameter	Description
filespec	Specify the name, and optionally, the path, of the file you want to open.
mode	A string indicating how you want to access the file. The mode can be one of the following: **"R"** (Read mode): opens the file for reading, starting at the beginning. **"W"** (Write mode): opens the file for writing. ***Warning!*** When you specify Write mode, any existing content in the file is discarded. In Update mode, any write operation clears the file of all data that follows the position you set. **Note.** Currently, the effect of the Update mode and the GetPosition and SetPosition methods is not well defined for Unicode files. Use the Update mode only on files stored with the ANSI character set. **"E"** (Conditional "exist" read mode): opens the file for reading only if it exists, starting at the beginning. If it doesn't exist, **Open** has no effect. Before attempting to read from the file, use the IsOpen property to confirm that it's open. **"N"** (Conditional "new" write mode): opens the file for writing, only if it doesn't already exist. If a file by the same name already exists, **Open** has no effect. Before attempting to write to the file, use the IsOpen property to confirm that it's open. You can insert an asterisk (*****) in the filename to ensure that a new file is created. The system replaces the asterisk with numbers starting at **1** and incrementing by 1, and checks for the existence of a file by each resulting name in turn. It uses the first name for which a file *doesn't* exist. In this way you can generate a set of automatically numbered files. If you insert more than one asterisk, all but the first one are discarded.
charset	A string indicating the character set you expect when you read the file, or the character set you want to use when you write to the file. You can abbreviate ANSI to "A" and Unicode to "U". All other character sets must be spelled out in full, for example, ASCII, THAI, or UTF8. If you don't specify a character set, the default value is ANSI. You can also use a record field value to specify the character set (for example, RECORD.CHARSET.) **Note.** If you attempt to read data from a file using a different character set than was used to write that data to the file, the methods used generate a runtime error or the data returned is unusable.
pathtype	If you have prepended a path to the file name, use this parameter to specify whether the path is an absolute or relative path. The valid values for this parameter are: • %FilePath_Relative (default) • %FilePath_Absolute If you don't specify *pathtype* the default is %FilePath_Relative. If the path is an absolute path, whatever path you specify is used verbatim. You must specify a drive letter as well as the complete path. You can't use any wildcards when specifying a path. The function automatically converts platform-specific separator characters to the appropriate form for where your PeopleCode program is executing. On a WIN32 system, UNIX "/" separators are converted to "\", and on a UNIX system, WIN32 "\" separators are converted to "/".

Note. The syntax of the file path does *not* depend on the file system of the platform where the file is actually stored; it depends only on the platform where your PeopleCode is executing.

Returns

A file object.

Example

The following example opens an existing Unicode file for reading:

```
&MYFILE = GetFile(&SOMENAME, "E", "U");
If &MYFILE.IsOpen Then
   while &MYFILE.ReadLine(&SOMESTRING);
      /* Process the contents of each &SOMESTRING */
   End-While;
   &MYFILE.Close();
End-If;
```

The following example opens a numbered file for writing in ANSI format, without overwriting any existing files:

```
&MYFILE = GetFile("c:\temp\item*.txt", "N", %FilePath_Absolute);
If &MYFILE.IsOpen Then
   &MYFILE.WriteLine("Some text.");
   &MYFILE.Close();
End-If;
```

The following example uses the CHARSET field to indicate the character set to be used:

```
&MYFILE = GetFile("INPUT.DAT", "R", RECORD.CHARSET);
```

See Also

Chapter 1, PeopleCode Built-in Functions, FileExists , page 184, Chapter 1, PeopleCode Built-in Functions, FindFiles , page 189, Chapter 1, PeopleCode Built-in Functions, GetFile , page 243, Chapter 13, File Class , page 873, Chapter 13, File Class , Open, page 882, Chapter 13, File Class , GetPosition, page 882, Chapter 13, File Class , SetPosition, page 890, Chapter 13, File Class , IsOpen, page 900, Chapter 13, File Class , File Access Interruption Recovery , page 875.

GetGrid

Syntax

GetGrid(**PAGE**.*pagename*, *gridname* [, *occursnumber*])

Description

The **GetGrid** function instantiates a grid object from the Grid class, and populates it with the grid specified by *gridname*, which is the Page Field Name on the General tab of that grid's page field properties.

Specify a grid name consisting of any combination of uppercase letters, digits and "#", "$", "@", and "_".

Note. PeopleSoft builds a page grid one row at a time. Because the Grid class applies to a complete grid, you can't attach PeopleCode that uses the Grid class to events that occur before the grid is built; the earliest event you can use is the page Activate Event.

See *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Activate Event.

Using the Occurs Count

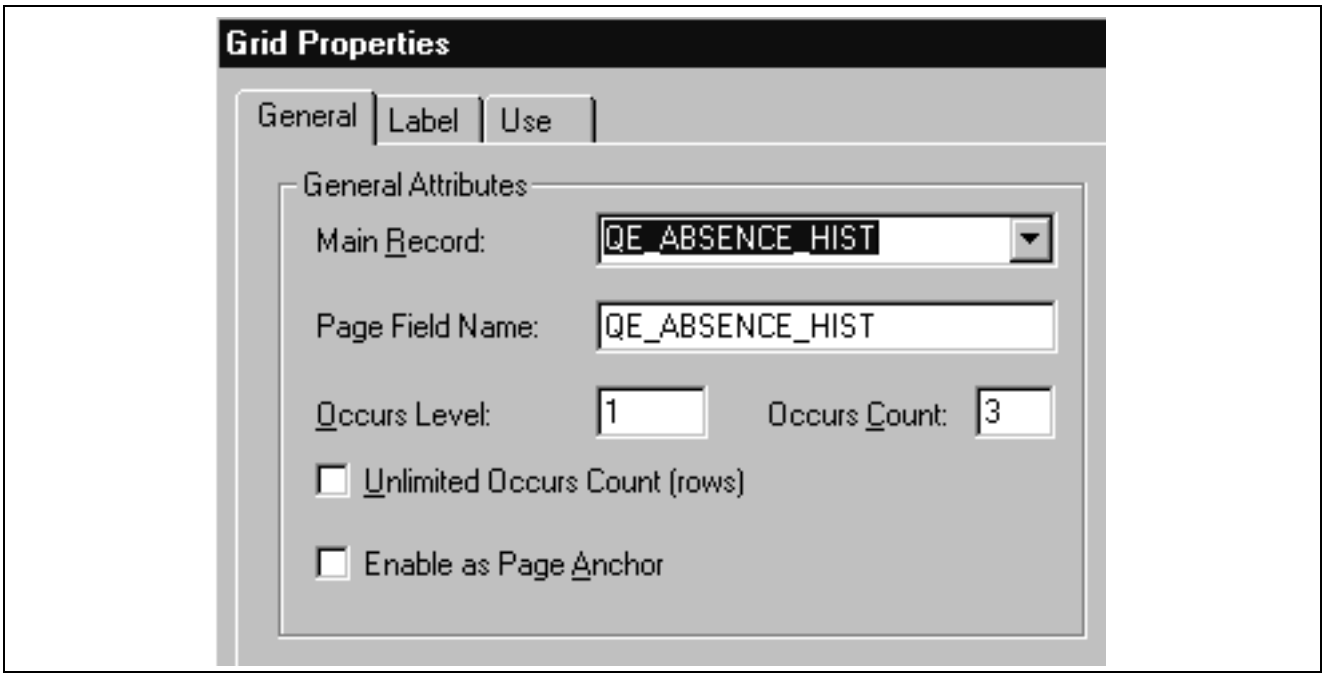
The occurs number (*occursnumber*) parameter that you specify with the GetGrid function does *not* refer to the row number of the grid. Rather, it's the occurrence of the grid within a scroll. If your grid is contained within a scroll, and the occurs number of the scroll is greater than 1, then to specify the correct grid within the scroll you must use the *occursnumber* parameter.

Using the Grid Name

When you place a grid on a page, the gird is automatically named the same as the name of the primary record of the scroll for the grid (in the Page Field Name.)

Note. If the name of the record changes, the Page Field Name is *not* automatically updated. You must change this name if you want the name of the grid to reflect the name of the record.

This is the name you use with the GetGrid function. You can change this name on the Record tab of the Grid properties.



Changing a grid name

To change a grid name:

1. Open the page in PeopleSoft Application Designer, select the grid and access the page field properties.
2. On the General tab, type the new grid name in **Page Field Name.**

Note. Every grid on a page must have a unique name.

Parameters

Parameter	Description
PAGE. *pagename*	Specify the name of the page definition containing the grid you want.
gridname	Specify the Page Field Name on the General tab of the grid's page field properties.
occursnumber	Optionally, specify the occurs number to identify the grid. Because you can use occurs count to generate multiple instances of a grid within a scroll just as you can with other page fields, *occursnumber* enables you to identify which one of those grids you want. The default value is 1.

Returns

A Grid object populated with the requested grid.

Example

This example retrieves the second grid named "EMPL_GRID" within a scroll:

```
local Grid &MYGRID;

&MYGRID = GetGrid(PAGE.EMPLOYEE_CHECKLIST, "EMPL_GRID", 2);
```

See Also

Chapter 14, Grid Classes , GetColumn, page 924.

GetHTMLText

Syntax

GetHTMLText(**HTML.***textname* [*, paramlist*])

Where *paramlist* is an arbitrary-length list of values of undetermined (Any) data type in the form: *inval1* [*, inval2*] ...

Description

The **GetHTMLText** function retrieves a predefined HTML text from an HTML definition. If any values are included in *paramlist*, they are substituted into the HTML text based on positional reference (for example, `%BIND(:1)` is the first parameter, `%BIND(:2)` is the second, and so on.)

Note. Use the GetHTMLText function only to retrieve HTML, or HTML that contains a JavaScript program, from an HTML definition. If you have an HTML definition that contains *only* JavaScript, use the **GetJavaScriptURL** Response object method to access it.

See Chapter 15, Internet Script Classes (iScript) , GetJavaScriptURL, page 952.

Restrictions on Use

Use this function with the PeopleSoft Internet Architecture. If run from a 2 tier environment, the parameter substitution does *not* take place.

Parameters

Parameter	Description
HTML. *textname*	Specify the name of an existing HTML text from an HTML definition.

Returns

The resulting HTML text is returned as a string.

Example

The following is the text in the HTML definition TEST_HTML:

```
This is a %BIND(:1) and %BIND(:2) test.
```

The following is the PeopleCode program:

```
Local Field &HTMLfield;

&string = GetHTMLText(HTML.TEST_HTML, "good", "simple");
&HTMLfield = GetRecord(Record.CHART_DATA).HTMLAREA;
&HTMLfield.Value = &string;
```

The output from &string (displayed in an HTML area control) is:

```
This is a good and simple test.
```

See Also

Chapter 15, Internet Script Classes (iScript) , page 931, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using HTML Definitions and the GetHTMLText Function, *PeopleSoft Application Designer*, Using Page Controls, Using HTML Areas.

GetImageExtents

Syntax

GetImageExtents(**IMAGE.***ImageName*)

Description

The **GetImageExtents** function returns the width and height of the image specified with *ImageName*.

Parameters

Parameter	Description
ImageName	Specify the name of the image on the page. This image must exist on the page.

Returns

An array of data type number, where element 1 is the image height and element 2 is the image width.

Example

```
Local array of number &ImageExtents;

&ImageExtents = GetImageExtents(Image.PT_TREE_EXPANDED);

WinMessage("Height is " | &ImageExtents[1] | " and width is " | &ImageExtents[2]);
```

See Also

PeopleSoft Application Designer, Creating Field Definitions, Image Field Type.

GetInterlink

Syntax

GetInterlink(**Interlink.***name*)

Description

The **GetInterlink** function instantiates a Business Interlink definition object based on a Business Interlink definition created in PeopleSoft Application Designer. The Business Interlink object can provide a gateway for PeopleSoft applications to the services of any external system.

After you use this function, you may want to refresh your page. The **Refresh** method, on a rowset object, reloads the rowset (scroll) using the current page keys. This causes the page to be redrawn**. GetLevel0().Refresh()** refreshes the entire page. If you only want a particular scroll to be redrawn, you can refresh just that part.

Generally, you do not use the **GetInterlink** function in a program you create from scratch. If you drag a Business Interlink definition from the project workspace (in PeopleSoft Application Designer) to an open PeopleCode editor window, a "template" is created, with values filled in based on the Business Interlink definition you dragged in.

The following is the template created from dragging the Business Interlink definition LDAP_SEARCHBIND to an open PeopleCode editor window.

```
/* ===>
   This is a dynamically generated PeopleCode template to be used only as a helper⇒
 to the application developer.  You need to replace all references to '<*>' OR⇒
 default values with  references to PeopleCode variables and/or a Rec.Fields.*/

/* ===> Declare and instantiate: */
Local Interlink &LDAP_SEARCHBI_1;
Local BIDocs &inDoc;
Local BIDocs &outDoc;
Local Boolean &RSLT;
Local number  &EXECRSLT;
&LDAP_SEARCHBI_1 =  GetInterlink(INTERLINK.LDAP_SEARCHBIND);

/* ===> You can use the following assignments to set the configuration parameters.
*/

&LDAP_SEARCHBI_1.Server = "jtsay111198.peoplesoft.com";
&LDAP_SEARCHBI_1.Port = 389;
```

```
&LDAP_SEARCHBI_1.User_DN = "cn=Admin,o=PeopleSoft";
&LDAP_SEARCHBI_1.Password = &password;
&LDAP_SEARCHBI_1.UserID_Attribute_Name = "uid";
&LDAP_SEARCHBI_1.URL = "file://psio_dir.dll";
&LDAP_SEARCHBI_1.BIDocValidating = "Off";


/* ===> You might want to call the following statement in a loop if there is more⇒
 than one row of data to be added.  */

/* ===> Add inputs: */
&inDoc = &LDAP_SEARCHBI_1.GetInputDocs("");
&ret = &inDoc.AddValue("User_ID", <*>);
&ret = &inDoc.AddValue("User_Password", <*>);
&ret = &inDoc.AddValue("Connect_DN", <*>);
&ret = &inDoc.AddValue("Connect_Password", <*>);
&Directory_Search_ParmsDoc = &inDoc.AddDoc("Directory_Search_Parms");
&ret = &Directory_Search_ParmsDoc.AddValue("Host", <*>);
&ret = &Directory_Search_ParmsDoc.AddValue("Port", <*>);
&ret = &Directory_Search_ParmsDoc.AddValue("Base", <*>);
&ret = &Directory_Search_ParmsDoc.AddValue("Scope", <*>);
&ret = &Directory_Search_ParmsDoc.AddValue("Filter", <*>);


/* ===> The following statement executes this instance: */
&EXECRSLT = &LDAP_SEARCHBI_1.Execute();
If ( &EXECRSLT <> 1 ) Then
   /* The instance failed to execute */
Else
&outDoc = &LDAP_SEARCHBI_1.GetOutputDocs("");
&ret = &outDoc.GetValue("Distinguished_Name", <*>);
&ret = &outDoc.GetValue("return_status", <*>);
&ret = &outDoc.GetValue("return_status_msg", <*>);

End-If; /* If NOT &RSLT ...  */
```

See Also

PeopleTools PeopleBook: PeopleSoft Business Interlink Application Developer Guide, "Generating the PeopleCode Template"

Parameters

Parameter	Description
Interlink. *name*	Specify the name of the Business Interlink definition from which to instantiate a Business Interlink object.

Returns

A Business Interlink object.

Example

The following example instantiates a Business Interlink object based on the Business Interlink definition QE_RP_SRAALL.

```
Local Interlink &SRA_ALL_1;

&SRA_ALL_1 = GetInterlink(Interlink.QE_RP_SRAALL);
```

See Also

Chapter 8, Business Interlink Class , page 659.
PeopleTools PeopleBook: PeopleSoft Business Interlink Application Developer Guide

GetJavaClass

Syntax

GetJavaClass(*ClassName*)

Description

The **GetJavaClass** function finds a Java class you can manipulate in PeopleCode. This is used for those classes that have static members, where it isn't appropriate to instantiate an object of the class. You can call only static methods, that is, class methods, with the object created with this function.

In Java, you access such static members of a class by using the class name:

```
result = java.class.name.SomeStaticMethod();
```

To do this in PeopleCode, do the following:

```
&Result = GetJavaClass("java.class.name").SomeStaticMethod();
```

Note. If you create a class that you want to call using GetJavaClass, it must be located in a directory specified in the PS_CLASSPATH environment variable.

Parameters

Parameter	Description
ClassName	Specify the name of an already existing class. This parameter takes a string value.

Returns

A JavaObject that refers to the named Java class.

Example

The Java class **java.lang.reflect.Array** has no public constructors and has only static methods. The methods are used to manipulate Java array objects. One of these static methods is GetInt:

```
public static int getInt(Object array, int index)
```

To use this method, get the class by using GetJavaClass. This code illustrates accessing the value of the fifth element of an integer array.

```
Local JavaObject &RefArray, &MyArray;

. . .

&RefArray = GetJavaClass("java.lang.reflect.Array");

. . .

&MyArray = CreateJavaArray(“int[]”, 24);

. . .

&FifthElement = &RefArray.getInt(&MyArray, 4);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateJavaArray , page 98, Chapter 1, PeopleCode Built-in Functions, CreateJavaObject , page 99, Chapter 16, Java Class, page 959.

GetLevel0

Syntax

GetLevel0()

Description

GetLevel0 creates a rowset object that corresponds to level 0 of the component buffer. If used from PeopleCode that isn’t associated with a page, it returns the base rowset from the current context.

Parameters

GetLevel0 has no parameters. However, it does have a default method, GetRow, and a shortcut. Specifying `GetLevel0()(1)` is the equivalent of specifying `GetLevel0().GetRow(1)`.

Returns

This function returns a rowset object that references the base rowset. For a component, this is the level 0 of the page. For a PeopleSoft Application Engine program, this is the state record rowset. For a message, this is the base rowset.

Note. You can also get the base rowset for a message using the GetRowset message class method, that is, `&MSG.GetRowset()`.

Example

The following code sample returns the level one rowset.

```
Local Rowset &ROWSET;

&ROWSET = GetLevel0().GetRow(1).GetRowset(SCROLL.LEVEL1_REC);
```

The following is equivalent to the previous example.

```
Local Rowset &ROWSET;

&ROWSET = GetLevel0()(1).GetRowset(SCROLL.LEVEL1_REC);
```

To reference a level 2 rowset you would have code similar to this:

```
Local Rowset &ROWSET_LEVEL2, &ROWSET_LEVEL0, &ROWSET_LEVEL1;

&ROWSET_LEVEL2 = GetLevel0().GetRow(1).GetRowset(SCROLL.LEVEL1_REC).GetRow(5). ⇒
 GetRowset(SCROLL.LEVEL2_REC);

  /* or */

&ROWSET_LEVEL0 = GetLevel0();
&ROWSET_LEVEL1 = &ROWSET_LEVEL0.GetRow(1).GetRowset(SCROLL.LEVEL1_REC);
&ROWSET_LEVEL2 = &ROWSET_LEVEL1.GetRow(5).GetRowset(SCROLL.LEVEL2_REC);
   /* or */
&ROWSET_LEVEL2 = GetLevel0()(1).LEVEL1_REC(5).GetRowset(SCROLL.LEVEL2_REC);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268, Chapter 27, Rowset Class , page 1479, *PeopleCode Developer's Guide*, Data Buffer Access.

GetMethodNames

Syntax

GetMethodNames(*Type, Name*)

Description

The **GetMethodNames** function returns either the method names for a Component Interface, or the function names of a WEBLIB record.

Parameters

Parameter	Description
Type	Specify the type of methods or functions you want returned. This parameter takes a string value. The values are: • WebLib • CompIntfc
Name	Specify the name of the Component Interface or WEBLIB record that you want to know the methods or functions for.

Returns

An array of string containing the method or function names.

Example

```
Local array of string &Array;

&Array = GetMethodNames("CompIntfc", CompIntfc.USER_PROFILE);

&Array = GetMethodNames("WebLib", Record.WEBLIB_PORTAL);
```

See Also

Chapter 10, Component Interface Classes , page 773, Chapter 15, Internet Script Classes (iScript) , Web Libraries , page 933.

GetMessage

Syntax

GetMessage()

Description

The **GetMessage** function is used in any of the messaging PeopleCode events. It retrieves a message from the message queue for the current message being processed. It creates and loads a data tree for the default message version, and returns NULL if not successful.

Parameters

None.

Returns

A reference to a message object if successful, NULL if not successful.

Example

```
Local message &MSG;
&MSG = GetMessage();
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateMessage , page 100, Chapter 17, Message Class , page 973.

GetMessageInstance

Syntax

GetMessageInstance(*pub_id, pub_nodename, channelname*)

Description

The **GetMessageInstance** function gets a message from the message queue.

Note. This function has been deprecated and is no longer supported.

GetMessageXmlDoc

Syntax

GetMessageXmlDoc()

Description

The **GetMessageXmlDoc** function is used in any of the messaging PeopleCode events. It retrieves an XML message, either from the message queue for asynchronous messages, or in memory for synchronous messages, for the current message being processed. An XML message is a message that is unstructured, that is, isn't based on a record hierarchy. It creates and loads a data tree for the default message version, and returns NULL if not successful.

Parameters

None.

Returns

A reference to an XmlDoc object if successful, NULL if not successful.

Example

The following example uses the GetMessageXmlDoc built-in function.

```
Local XmlDoc &BIGMAN;
Local XmlNode &node, &root;
Local string &outstring;
Local Rowset &LEVEL0;
Local Record &SALES_ORDER_INFO, &REC;

&CRLF = Char(13) | Char(10);

&BIGMAN = GetMessageXmlDoc();

&root = &BIGMAN.DocumentElement;
&child_count = &root.ChildNodeCount;

/* Get values out of XMLDoc */
&node_array = &root.GetElementsByTagName("QE_ACCT_ID");
&acct_id_node = &node_array.Get(2);
&account_id_value = &acct_id_node.NodeValue;

&node_array = &root.GetElementsByTagName("QE_ACCOUNT_NAME");
&acct_name_node = &node_array.Get(2);
&account_name_value = &acct_name_node.NodeValue;

&node_array = &root.GetElementsByTagName("QE_ADDRESS");
&address_node = &node_array.Get(2);
&address_value = &address_node.NodeValue;

&node_array = &root.GetElementsByTagName("QE_PHONE");
&phone_node = &node_array.Get(2);
&phone_value = &phone_node.NodeValue;

&outstring = "GetMessageXMLDoc Test";
&outstring = &outstring | &CRLF | &account_id_value | &CRLF | &account_name_value⇒
```

```
 | &CRLF | &address_value | &CRLF | &phone_value;

&SALES_ORDER_INFO = CreateRecord(Record.QE_SALES_ORDER);
&SALES_ORDER_INFO.GetField(Field.QE_ACCT_ID).Value = &account_id_value;
&SALES_ORDER_INFO.GetField(Field.DESCRLONG).Value = &outstring;
&SALES_ORDER_INFO.Update();
```

See Also

Chapter 1, PeopleCode Built-in Functions, PublishXmlDoc , page 352, Chapter 1, PeopleCode Built-in Functions, SyncRequestXmlDoc , page 455, Chapter 34, XmlDoc Class , page 1757, Chapter 17, Message Class , page 973.

GetNextNumber

Syntax

GetNextNumber ({*record.field* | *record_name*, *field_name*}, *max_number*)

Description

The **GetNextNumber** function increments the value in a record for the field you specify by one and returns that value. You might use this function to increment an employee ID field by one when you are adding a new employee. If the new value generated exceeds *max_number*, a negative value is returned and the field value isn't incremented. The maximum value possible for *max_number* is 2147483647.

PeopleCode Event Considerations

Because this function results in a database update (specifically, UPDATE, INSERT, and DELETE) it should only be issued in the following events:

- SavePreChange
- WorkFlow
- SavePostChange

If you use this function in an event other than these, you need to ensure that the dataflow is correct and that you don't receive unexpected results.

GetNextNumber and GetNextNumberWithGaps

The following is some of the differences between the two functions, to enable you to better chose which one is better for your application.

GetNextNumber	GetNextNumberWithGaps
No AutoCommit (which can be a problem, as the table is locked until all Save events are finished.)	AutoCommit (this can be a performance enhancement as table is not locked as long
	Ability to use WHERE criteria for maintaining multiple sequence numbers in a single record

GetNextNumber	GetNextNumberWithGaps
	Ability to increment by more than 1.
Allowed in SavePostChange	

Parameters

Parameter	Description
record.field	Specify the record and field identifiers for the field for which you want the new number. This is the recommended way to identify the field.
record_name	Specify as a string the name of the record containing the field for which you want the new number. This parameter with *field_name* was used prior to PeopleTools 8.
field_name	Specify as a string the name of the field for which you want the new number. This parameter with *record_name* was used prior to PeopleTools 8. **Note.** If you use the older syntax (*record_name*, *field_name*), you have to manually update these two parameters in your programs whenever that record or field is renamed. The new syntax (*record.field)* is automatically updated, so you won't have to maintain it.
max_number	Specify the highest allowed value for the field you're incrementing. The maximum value possible for *max_number* is 2147483647.

Returns

A Number value equal to the highest value of the field specified plus one.

GetNextNumber returns an error if the value to be returned would be greater than *max_number*. The function returns one of the following:

Numeric Value	Constant Value	Description
Number		The new number
-1	%GetNextNumber_SQLFailure	SQL failure
-2	%GetNextNumber_TooBig	Number too large, beyond *max_number*
-3	%GetNextNumber_NotFound	No number found, invalid data format

Example

```
If %Component = "RUN_AR33000" Then
   DUN_ID_NUM = GetNextNumber(INSTALLATION_AR.DUN_ID_NUM, 99999999);
End-if;
```

The following uses the constant to check for the value returned:

```
&VALUE = GetNextNumber(INSTALLATION_AR.DUN_ID_NUM, 999);

Evaluate &VALUE
When = %GetNextNumber_SQLFailure
```

```
     /* do processing */
When = %GetNextNumber_TooBig
     /* do processing */
When = %GetNextNumber_NotFound
     /* Do processing */
When-other
     /* do other processing */
End-Evaluate;
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetNextNumberWithGaps , page 259.

GetNextNumberWithGaps

Syntax

GetNextNumberWithGaps(*record.field*, *max_number*, *increment* [, *WHERE_Clause, paramlist*])

Where *paramlist* is an arbitrary-length list of values in the form:
var1 [*, var2*] ...

Description

The **GetNextNumberWithGaps** function determines the highest value in a table for the field you specify, and returns that value plus *increment*. This function also enables you to specify a SQL WHERE clause as part of the function for maintaining multiple sequence numbers in a single record.

Note. GetNextNumberWithGaps also issues a COMMIT after incrementing the sequence number if no other database updates have occurred since the last COMMIT. This limits the time a database lock is held on the row and so may improve performance.

PeopleCode Event Considerations

Because this function results in a database update (specifically, UPDATE, INSERT, and DELETE) it should only be issued in the following events:

- SavePreChange
- WorkFlow

If you use this function in an event other than these, you need to ensure that the dataflow is correct and that you don’t receive unexpected results.

Parameters

Parameter	Description
record.field	Specify the record and field identifiers for the field for which you want the new number. This is the recommended way to identify the field.
max_number	Specify the highest allowed value for the field you're incrementing. You can specify up to 31 digits for this value.
increment	Specify the value you want the numbers incremented by. You can specify up to 31 digits for this value.
WHERE_Clause	Specify a WHERE clause for maintaining multiple sequence numbers.
paramlist	Parameters for the WHERE clause.

Returns

A Number value equal to the highest value of the field specified plus one.

GetNextNumberWithGaps returns an error if the value to be returned would be greater than *max_number*. The function returns one of the following:

Numeric Value	Constant Value	Description
Number		The new number
-1	%GetNextNumber_SQLFailure	SQL failure
-2	%GetNextNumber_TooBig	Number too large, beyond *max_number*
-3	%GetNextNumber_NotFound	No number found, invalid data format

Example

The following PeopleCode:

```
&greg = GetNextNumberWithGaps(GREG.DURATION_DAYS, 999999, 50, "where emplid = :1",⇒
 8001);
```

results in the following:

```
2-942   21.53.09    0.000 Cur#4.PTTST81B RC=0 Dur=0.000 Connect=PTTST81B/sa/
2-943   21.53.09    0.000 Cur#4.PTTST81B RC=0 Dur=0.000 COM Stmt=UPDATE PS_GREG⇒
 SET DURATION_DAYS = DURATION_DAYS + 50 where emplid = 8001
2-944   21.53.09    0.000 Cur#4.PTTST81B RC=0 Dur=0.000 EXE
2-945   21.53.09    0.000 Cur#4.PTTST81B RC=0 Dur=0.000 COM Stmt=SELECT ⇒
DURATION_DAYS FROM PS_GREG where emplid = 8001
2-946   21.53.09    0.000 Cur#4.PTTST81B RC=0 Dur=0.000 EXE
2-947   21.53.09    0.000 Cur#4.PTTST81B RC=0 Dur=0.000 Fetch
2-948   21.53.09    0.010 Cur#4.PTTST81B RC=0 Dur=0.010 Commit
2-949   21.53.09    0.010 Cur#4.PTTST81B RC=0 Dur=0.010 Disconnect
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetNextNumber , page 257.

GetNextProcessInstance

Syntax

GetNextProcessInstance([*Commit*])

Description

The **GetNextProcessInstance** function retrieves the next available process instance from the Process Scheduler System table. When determining to find the next process instance in the sequence, the function ensures the next available process instance does not exist in both the Process Request and Message Log tables.

By default, the function commits the changes to the Process Scheduler System table to set it to the next available process instance for the next available request. If this function is called within a PeopleCode function where issuing a COMMIT to the database destroys a unit of work, specify "0" for *Commit*.

Parameters

Parameter	Description
Commit	Specify whether the current data instance should be committed to the database. This parameter takes a string value: "1", commit the data, "0", do not commit the data. "1" is the default value.

Returns

Return a number: 0 if successful, 1 if the system detected an error retrieving the next available process instance.

See Also

Chapter 23, ProcessRequest Class , page 1269, .

GetNRXmlDoc

Syntax

GetNRXmlDoc(*NRID, EntityName*)

Description

The **GetNRXmlDoc** function is used in any of the messaging PeopleCode events. It retrieves an XML message, categorized as non-repudiation, from the message queue for the specified non-repudiation ID. An XML message is a message that is unstructured, that is, isn't based on a record hierarchy. It creates and loads a data tree for the default message version, and returns Null if not successful.

Parameters

Parameter	Description
NRID	Specify the non-repudiation ID for the XML message that you want to retrieve. This parameter takes a numeric value.
EntityName	Specify the name of the entity that signed the data, as a string. For Peoplesoft, this is the node name.

Returns

A reference to an XmlDoc object if successful, Null if not successful.

See Also

Chapter 1, PeopleCode Built-in Functions, GetMessageXmlDoc , page 255, Chapter 34, XmlDoc Class , page 1757.

GetPage

Syntax

GetPage(**PAGE**.*pagename*)

Description

The **GetPage** function returns a reference to a page object. Generally, page objects are used to hide or unhide pages in a component.

Generally, the PeopleCode used to manipulate a page object would be associated with PeopleCode in the Activate event.

Note. The page object shouldn't be used until after the Component Processor has loaded the page: that is, don't instantiate this object in RowInit PeopleCode, use it in PostBuild or Activate instead.

Note

An expression of the form

```
PAGE.name.property
```

is equivalent to **GetPage**(*name*).*property*.

Parameters

Parameter	Description
PAGE. *pagename*	The name of the page for which you want to create an object reference. Must be a page in the current context.

Returns

A page object that references the page.

Example

In the following example, a page is hidden based on the value of the current field.

```
If PAYROLE_TYPE = "Global" Then
   GetPage(PAGE.JOB_EARNINGS).Visible = False;
End-If;
```

See Also

Chapter 20, Page Class , page 1089, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Understanding Current Context.

GetPubContractInstance

Syntax

GetPubContractInstance(*pub_id*, *pub_nodename*, *channelname*, *sub_nodename*)

Description

The **GetPubContractInstance** function gets a message from the message queue. It creates and loads a data tree for the default message version, and returns NULL if not successful. This function is used for publication contract error correction when the error correction process needs to fetch a particular message instance for the publication contract in error. SQL on the Publication Contract table is used to retrieve the key fields.

Note. This function should *not* be used in standard message processing. It should be used only when correcting or debugging a publication contract that is in error.

Parameters

Parameter	Description
pub_id	Specifies the PubID of the message.
pub_nodename	Specifies the PubNodeName of the message.
ChannelName	Specifies the channel name of the message.
sub_nodename	Specifies the subscribing node of the message.

Returns

A reference to a message object if successful, NULL if not successful.

Example

```
Local message &MSG;

&MSG = GetPubContractInstance(&PUBID, &PUBNODE, &CHANNEL, &SUBNODENAME);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateMessage , page 100, Chapter 1, PeopleCode Built-in Functions, GetMessage , page 255, Chapter 1, PeopleCode Built-in Functions, GetSubContractInstance , page 274, Chapter 1, PeopleCode Built-in Functions, GetMessageInstance , page 255, Chapter 17, Message Class , page 973.

GetPubHeaderXmlDoc

Syntax

GetPubHeaderXmlDoc(*PubID, PubNode, ChannelName, VersionName*)

Description

The message header, also known as the message instance, is the published message *before* any transformations were performed. The **GetPubHeaderXmlDoc** function gets the message header from the message queue.

Note. This function should *not* be used in standard message processing. It should only be used when correcting or debugging a publication contract that is in error.

Parameters

Parameter	Description
PubID	Specify the PubID of the message.
PubNode	Specify the Pub Node Name of the message.
ChannelName	Specify the channel name of the message.
VersionName	Specify the version name of the message.

Returns

A reference to an XmlDoc object if successful, NULL if not successful.

See Also

Chapter 1, PeopleCode Built-in Functions, ReSubmitPubHeaderXmlDoc , page 374, Chapter 1, PeopleCode Built-in Functions, GetPubXmlDoc , page 264.

GetPubXmlDoc

Syntax

GetPubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName, SubNode*)

Description

The **GetPubXmlDoc** function gets a message from the message queue. This is the message *after* any transformations have been preformed. It creates and loads a data tree for the specified message version, and returns NULL if not successful. This function is used for publication contract error correction when the error correction process needs to fetch a particular message instance for the publication contract in error. SQL on the Publication Contract table is used to retrieve the key fields.

Note. This function should *not* be used in standard message processing. It should only be used when correcting or debugging a publication contract that is in error.

Parameters

Parameter	Description
PubID	Specify the PubID of the message.
PubNode	Specify the Pub Node Name of the message.
ChannelName	Specify the channel name of the message.
VersionName	Specify the version name of the message.
MessageName	Specify the name of the message.
SubNode	Specify the subnode of the message.

Returns

A reference to an XmlDoc object if successful, NULL if not successful.

See Also

Chapter 1, PeopleCode Built-in Functions, ReSubmitPubXmlDoc , page 375, Chapter 1, PeopleCode Built-in Functions, ReSubmitPubHeaderXmlDoc , page 374.

GetRecord

Syntax

```
GetRecord([RECORD.recname])
```

Description

GetRecord creates a reference to a record object for the current context, that is, from the row containing the currently executing program.

The following code:

```
&REC = GetRecord();
```

is equivalent to:

```
&REC = GetRow().GetRecord(Record.recname);
```

or

```
&REC = GetRow().recname;
```

Note. This function is invalid for PeopleCode programs located in events that aren't associated with a specific row and record at the point of execution. That is, you cannot use this function in PeopleCode programs on events associated with high-level objects like pages (the *Activate* event) or components (component events).

Parameters

With no parameters, this function returns a record object for the current context (the record containing the program that is running).

If a parameter is given, **RECORD.** *recname* must specify a record in the current row.

Returns

GetRecord returns a record object.

Example

In the following example, the level 2 rowset (scroll) has two records: EMPL_CHKLST_ITM, (the primary record) and CHKLST_ITM_TBL. If the code is running from a field on the EMPL_CHKLST_ITM record, the following returns a reference to that record:

```
&REC = GetRecord(); /*returns primary record */
```

The following returns the other record in the current row.

```
&REC2 = GetRecord(RECORD.CHKLST_ITM_TBL);
```

The following event uses the @ symbol to convert a record name that's been passed in as a string to a component name.

```
Function set_sub_event_info(&REC As Record, &NAME As string)
   &FLAGS = CreateRecord(RECORD.DR_LINE_FLG_SBR);
   &REC.CopyFieldsTo(&FLAGS);
   &INFO = GetRecord(@("RECORD." | &NAME));
   If All(&INFO) Then
      &FLAGS.CopyFieldsTo(&INFO);
   End-If;
End-Function;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateRecord , page 105, Chapter 17, Message Class , page 973, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Understanding Current Context.

GetRelField

Syntax

GetRelField(*ctrl_field*, *related_field*)

Description

GetRelField retrieves the value of a related display field and returns it as an unspecified (Any) data type.

Note. This function remains for backward compatibility only. Use the **GetRelated** field class method instead.

The field *ctrl_field* specifies the display control field, and *related_field* specifies the name of the related display field whose value is to be retrieved. In most cases, you could get the value of the field by referencing it directly. However, there are two instances where **GetRelField** can be useful:

- If there are two related display fields bound to the same record field, but controlled by different display control fields, use this function to specify which of the two related display fields you want.

- If all of a page's level-zero fields are search keys, the Component Processor does not load the entire row of level-zero data into the component buffer; it only loads the search keys. Adding a non-search-key level-zero field to the page would cause the Component Processor to load the entire row into the component buffer. To prevent a large row of data from being loaded into the buffer, you may occasionally want to make a level-zero display-only field a related display, even though the field is in the primary level-zero record. You won't be able to reference this related display field directly, but you can using **GetRelField.**

See Chapter 12, Field Class , GetRelated, page 848.

Using GetRelField With a Control Field

PeopleCode events on the Control Field can be triggered by the Related Edit field. When this happens, there can be different behavior than with other types of fields:

- If the events are called from FieldEdit of the Control Field, and that FieldEdit is triggered by a change in the Related Edit field, the functions return the previous value.
- If the events are called from FieldChange of the Control Field, and that FieldChange is triggered by a change in the Related Edit field, the functions return the value entered into the Related Edit. This may be a partial value that will subsequently be expanded to a complete value when the processing is complete.

Example

In the following example, there are two related display fields in the page bound to PERSONAL_DATA.NAME. One is controlled by the EMPLID field of the high-level key, the other controlled by an editable DERIVED/WORK field in which the user can enter a new value. Use **GetRelField** to get the value of the related display controlled by EMPLID.

```
/* Use a related display of a required non-default field to verify
 * that the new Employee Id is not already in use */
If GetRelField(EMPLID, PERSONAL_DATA.NAME) <> "" Then
   Error MsgGet(1000, 65, "New Employee ID is already in use.  Please reenter.");
End-If;
```

See Also

PeopleCode Developer's Guide, Data Buffer Access.

GetRow

Syntax

GetRow()

Description

Use the **GetRow** function to obtain a row object for the current context, that is the row containing the currently executing program.

Using the GetRow function is equivalent to:

```
&ROW = GetRowset().GetRow(CurrentRowNumber());
```

Note. For PeopleCode programs located in events that are not associated with a specific row at the point of execution, this function is invalid. That is, you cannot use this function in PeopleCode programs on events associated with high-level objects like pages or components.

Parameters

None.

Returns

GetRow returns a row object that references the current row in the component buffers. If the program is not being run from a page (such as from PeopleSoft Application Engine, or as part of a Message program) it references that data.

Example

```
Local Row &ROW;

&ROW = GetRow();
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268, Chapter 26, Row Class , page 1465, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Understanding Current Context.

GetRowset

Syntax

GetRowset([**SCROLL.***scrollname*])

Description

Use the **GetRowset** function to get a rowset object based on the current context. That is, the rowset is determined from the row containing the program that is running.

Note

An expression of the form

```
RECORD.scrollname.property
```

or

```
RECORD.scrollname.method(. . .)
```

is converted to an object expression by using GetRowset(**SCROLL.** *scrollname*).

Parameters

If a parameter is specified, it must be the name of the primary record for the scroll that is a child of the current context.

Returns

With no parameters, **GetRowset** returns a rowset object for the rowset containing the currently running program. If a parameter is specified, it returns a rowset for that child scroll. *scrollname* must be the name of the primary record for the scroll.

Example

In the following example, RS1 is a level 1 rowset, and RS2 is a child rowset of RS1.

```
Local Rowset &RS1, &RS2;

&RS1 = GetRowset();
&RS2 = GetRowset(SCROLL.EMPL_CHKLST_ITM);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetLevel0 , page 253, Chapter 27, Rowset Class , page 1479, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Understanding Current Context.

GetSelectedTreeNode

Syntax

GetSelectedTreeNode(**RECORD.***recordname*)

Description

GetSelectedTreeNode determines what node the user has selected in a dynamic tree control.

Note. Dynamic tree controls have been deprecated. Use **GenerateTree** or Tree Viewer.

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateTree , page 230, Appendix B, Appendix B: Viewing Trees From Application Pages , page 1939.

GetSession

Syntax

GetSession()

Description

The **GetSession** function gets a PeopleSoft session object.

After you use GetSession, you can instantiate many other types of objects, like Component Interfaces, data trees, and so on.

After you use GetSession you must connect to the system using the **Connect** property.

If you are connecting to the existing session and not doing additional error checking, you may want to use the **%Session** system variable instead of GetSession. %Session returns a connection to the existing session.

Parameters

None.

Returns

A PeopleSoft session object.

Example

```
Local ApiObject &MYSESSION;

&MYSESSION = GetSession();
```

See Also

Chapter 10, Component Interface Classes , page 773, Chapter 21, PortalRegistry Classes , page 1093, Chapter 24, Query Classes , page 1303, Chapter 28, Search Classes , page 1509, Chapter 29, Session Class , page 1559, Chapter 33, Tree Classes , page 1653, Chapter 3, System Variables , %Session , page 577.

GetSetId

Syntax

GetSetId({**FIELD**.*fieldname* | *text_fieldname*}, *set_ctrl_fieldvalue*, {**RECORD**.*recname* | *text_recname*}, *treename*)

Description

GetSetId returns a string containing a setID based on a set control field (usually BUSINESS_ UNIT), a set control value, and one of the following:

- The name of a control table (or view) belonging to a record group in the TableSet Control controlled by the set control value.
- The name of a tree in the TableSet Control controlled by the set control value.

If you want to pass a control recordname to the function, you must pass an empty string in the *treename* parameter. Conversely, if you want to pass a treename, you must pass an empty string in the *text_recname* parameter. In practice, treenames are rarely used in this function.

Parameters

Parameter	Description
fieldname	Specify the set control field name as a FIELD reference. Use this parameter (recommended) or the *text_fieldname* parameter.
text_fieldname	Specify the name of the set control field as a string. Use this parameter or the *fieldname* parameter.
set_ctrl_fieldvalue	Specify the value of the set control field as a string.
recname	Specify as a RECORD reference the name of the control record belonging to the record group for which you want to obtain the setID corresponding to the set control value. Use this parameter (recommended) or the *text_recname* parameter.
text_recname	Specify as a string the name of the control record belonging to the record group for which you want to obtain the setID corresponding to the set control field value. Use this parameter or the *recname* parameter.
treename	Specify as a string the name of the tree for which you want to obtain the setID corresponding to the set control field value.

Returns

GetSetId returns a five-character setID string.

Example

In this example, BUSINESS_UNIT is the Set Control Field, and PAY_TRMS_TBL is a control table belonging to a record group controlled by the current value of BUSINESS_UNIT. The function returns the setID for the record group.

```
&SETID = GetSetId(FIELD.BUSINESS_UNIT, &SET_CTRL_VAL, RECORD.PAY_TRMS_TBL, "");
```

See Also

PeopleSoft Application Designer, Planning Records, Control Tables, and TableSets, Control Tables.

GetSQL

Syntax

GetSQL(**SQL**.*sqlname* [, *paramlist*])

Where *paramlist* is an arbitrary-length list of values in the form:
inval1 [, *inval2*] ...

Description

The **GetSQL** function instantiates a SQL object and associates it with the SQL definition specified by *sqlname*. The SQL definition must already exist, either created using PeopleSoft Application Designer or the **StoreSQL** function.

Processing of the SQL definition is the same as for a SQL statement created by the **CreateSQL** function.

Setting Data Fields to Null

This function does *not* set Component Processor data buffer fields to NULL after a row not found fetching error. However, it does set fields that aren't part of the Component Processor data buffers to NULL.

Using Arrays With paramlist

You can use a parameter of type "Array of Any" in place of a list of bind values. This is primarily used when you don't know the number of values being supplied until the code runs.

For example, suppose you had a SQL definition INSERT_TEST, that had PeopleCode that dynamically (that is, at runtime) generated the following SQL statement:

```
"INSERT INTO PS_TESTREC (TESTF1, TESTF2, TESTF3, TESTF4, . . .TESTN) VALUES (:1, ⇒
:2, %DateTimeIn(:3), %TextIn(:4). . .N)";
```

Suppose you have placed the values to be inserted into an Array of Any, say &AAny:

```
&AAny = CreateArrayAny("a", 1, %DateTime, "abcdefg", . . .N);
```

You can execute the insert by:

```
GetSQL(SQL.INSERT_TEST, &AAny);
```

Because the Array of Any promotes to absorb any remaining select columns, it must be the last parameter for the SQL object Fetch method or (for results) SQLExec. For binding, it must be the only bind parameter, as it is expected to supply all the bind values needed.

Parameters

Parameter	Description
SQL.*sqlname*	Specify the name of a SQL definition.
paramlist	Specify input values for the SQL string.

Returns

A SQL object.

Example

The following code creates and opens an SQL object on the SQL definition stored as ABCD_XY (for the current market, database type and as of date). It binds the given input values, and executes the statement. If the SQL.ABCD is a SELECT, this should be followed by a series of Fetch method calls.

```
&SQL = GetSQL(SQL.ABCD_XY, ABSENCE_HIST, &EMPLID);
```

The following is a generic function that can be called from multiple places to retrieve a specific record using the SQL Objects.

```
Local SQL &SQL;
Local string &SETID, &TEMPLATE;
Local date &EFFDT;

Function FTP_GET_TEMPLATE(&REC As Record) Returns Boolean ;
```

```
   &TEMPLATE = FTP_RULE_TEMPLATE;
   &EFFDT = EFFDT;
   &SETID = SETID;
   &SQL = GetSQL(SQL.FTP_TEMPLATE_SELECT, &SETID, &TEMPLATE, &EFFDT);
   If &SQL.Status = 0 Then
      If &SQL.Fetch(&REC) Then
         &SQL.Close();
         Return True;
      End-If;
   Else
      &TITLE = MsgGet(10640, 24, "SQL Error");
      MessageBox(64, &TITLE, 10640, 23, "SQL Object Not Found in SQL", ⇒
SQL.FTP_TEMPLATE_SELECT);
   End-If;
   &SQL.Close();
   Return False;
End-Function;
```

The SQL definition FTP_TEMPLATE_SELECT has the following code. Note that it uses the %List and %EFFDTCHECK meta-SQL statements. This makes the code easier to maintain: if there are any changes to the underlying record structure, this SQL definition *won't* have to change:

```
SELECT %List(FIELD_LIST,FTP_DEFAULT_TBL A)
FROM PS_FTP_TEMPLATE_TBL A
WHERE A.SETID = :1  AND A.FTP_RULE_TEMPLATE = :2
AND %EFFDTCHECK(FTP_DEFAULT_TBL A1,A,:3)  AND A.EFF_STATUS = 'A'
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109, Chapter 1, PeopleCode Built-in Functions, DeleteSQL , page 141, Chapter 1, PeopleCode Built-in Functions, FetchSQL , page 180, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442, Chapter 31, SQL Class , page 1613, Chapter 31, SQL Class , Open, page 1626.

GetStoredFormat

Syntax

GetStoredFormat(*scrollpath, target_row,*
[*recordname.*]*fieldname*)

where *scrollpath* is:
[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD.***target_recname*

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

GetStoredFormat returns the name of a field's custom stored format.

Note. This function remains for backward compatibility only. Use the **StoredFormat** Field class property instead.

To return the format for a field on level zero of the page, pass 1 in *target_row*.

See Also

Chapter 12, Field Class , StoredFormat, page 868, *PeopleCode Developer's Guide*, Data Buffer Access.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	An integer specifying the row of the target field. If you are testing a field on level zero, pass 1 in this parameter.
[*recordname* **.**]*fieldname*	The name of the field from which to get the stored format name. The field can be on any level of the active page. The *recordname* prefix is required if the call to **GetStoredFormat** is in a record definition other than *recordname*.

Returns

Returns a String equal to the name of the stored custom format for the field.

Example

This example returns a string containing the custom format for postal codes on level zero of the page or on the current row of scroll level one. This function is called in the RowInit event, so no looping is necessary.

```
Function get_postal_format() Returns string
   &CURR_LEVEL = CurrentLevelNumber();
   Evaluate &CURR_LEVEL
   When = 0
      &FORMAT = GetStoredFormat(POSTAL, 1);
   When = 1
      &FORMAT = GetStoredFormat(POSTAL, CurrentRowNumber(1));
   End-Evaluate;
   Return (&FORMAT);
End-Function;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetDisplayFormat , page 419, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

GetSubContractInstance

Syntax

GetSubContractInstance(*pub_id*, *pub_nodename*, *channelname*, *messagename*, *sub_name*)

Description

The **GetSubContractInstance** function gets a message from the message queue. It creates and loads a data tree for the default message version, and returns NULL if not successful. This function is used for subscription contract error correction, when the error correction process needs to fetch a particular message instance for the subscription contract in error. SQL on the Subscription Contract table is used to retrieve the key fields.

Note. This function should *not* be used in standard message processing. It should be used only when correcting or debugging a subscription contract that is in error.

This function automatically populates the error properties at the Message, Rowset, Row, Record, and Field object levels. That is:

- Message object (sets the **IsEditError** property to True if errors exist anywhere in this message)
- Rowset (sets the **IsEditError** property to True if errors exist anywhere in this rowset)
- Row (sets the **IsEditError** property to True if errors exist anywhere in this row)
- Record (sets the **IsEditError** property to True if errors exist anywhere in this record)
- Field (sets the **EditError** property to True if errors exist on this field, also sets the **MessageNumber** and **MessageSetNumber** properties)

See Also

Chapter 17, Message Class , IsEditError, page 1002, Chapter 27, Rowset Class , IsEditError, page 1506, Chapter 26, Row Class , IsEditError, page 1474, Chapter 25, Record Class , IsEditError, page 1462, Chapter 12, Field Class , EditError, page 854, Chapter 12, Field Class , MessageNumber, page 863, Chapter 12, Field Class , MessageSetNumber, page 863.

Parameters

Parameter	Description
pub_id	Specifies the PubID of the message.
ChannelName	Specifies the PubNodeName of the message.
MessageName	Specified the name of the message to get.
sub_name	Specified the subscription name (SubName) of the message.

Returns

A reference to a message object if successful, NULL if not successful.

Example

The following is a complete program for getting messages from the error queue and populating a page based on the information in the message.

```
Local Message &ITEM_MESSAGE;
Local Rowset &MASTER_ROWSET, &DETAIL_ROWSET;
Local Record &MASTER_REC, &MASTER_REC_WRK;
```

```
&ITEM_MESSAGE = GetSubContractInstance(PSAPMSGSUBCON_I.PUBID, PSAPMSGSUBCON_⇒
I.PUBNODE, PSAPMSGSUBCON_I.CHNLNAME, PSAPMSGSUBCON_I.MSGNAME, PSAPMSGSUBCON_⇒
I.SUBNAME);

&MASTER_ROWSET = &ITEM_MESSAGE.GetRowset();


For &I = 1 To &MASTER_ROWSET.RowCount
   /*
   &MASTER_REC = &MASTER_ROWSET(&I).MASTER_ITEM_ECB;
   &MASTER_REC_WRK = CreateRecord(RECORD.MASTER_ITEMEWRK);
   &MASTER_REC.CopyFieldsTo(&MASTER_REC_WRK);
*/
   MASTER_ITEMEWRK.SETID = &MASTER_ROWSET(&I).MASTER_ITEM_ECB.SETID.Value;
   MASTER_ITEMEWRK.PROCESS_INSTANCE = &MASTER_ROWSET(&I).MASTER_ITEM_ECB.PROCESS_⇒
INSTANCE.Value;
   MASTER_ITEMEWRK.INV_ITEM_ID = &MASTER_ROWSET(&I).MASTER_ITEM_ECB.INV_ITEM_⇒
ID.Value;
   &DETAIL_ROWSET = &MASTER_ROWSET(&I).GetRowset(1);
   For &J = 1 To &DETAIL_ROWSET.RowCount

      InsertRow(RECORD.INV_ITEMS_ECWRK, &J);

      UpdateValue(INV_ITEMS_ECWRK.SETID, &J, &DETAIL_ROWSET(&J).INV_ITEMS_EC_⇒
BK.SETID.Value);
      UpdateValue(INV_ITEMS_ECWRK.PROCESS_INSTANCE, &J, &DETAIL_ROWSET(&J).INV_⇒
ITEMS_EC_BK.PROCESS_INSTANCE.Value);
      UpdateValue(INV_ITEMS_ECWRK.INV_ITEM_ID, &J, &DETAIL_ROWSET(&J).INV_ITEMS_EC_⇒
BK.INV_ITEM_ID.Value);
      UpdateValue(INV_ITEMS_ECWRK.INV_ITEM_SIZE, &J, &DETAIL_ROWSET(&J).INV_ITEMS_⇒
EC_BK.INV_ITEM_SIZE.Value);
      UpdateValue(INV_ITEMS_ECWRK.EFFDT, &J, &DETAIL_ROWSET(&J).INV_ITEMS_EC_⇒
BK.EFFDT.Value);
   End-For;
   DeleteRow(RECORD.INV_ITEMS_ECWRK, &DETAIL_ROWSET.RowCount + 1);
End-For;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateMessage , page 100, Chapter 1, PeopleCode Built-in Functions, GetMessage , page 255, Chapter 1, PeopleCode Built-in Functions, GetPubContractInstance , page 263, Chapter 1, PeopleCode Built-in Functions, GetMessageInstance , page 255, Chapter 17, Message Class , page 973.

GetSubXmlDoc

Syntax

GetSubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName*)

Description

The **GetSubXmlDoc** function gets a message from the message queue. It creates and loads a data tree for the specified message version, and returns NULL if not successful. This function is used for subscription contract error correction, when the error correction process needs to fetch a particular message instance for the subscription contract in error. SQL on the Subscription Contract table is used to retrieve the key fields.

Note. This function should *not* be used in standard message processing. It should only be used when correcting or debugging a subscription contract that is in error.

Parameters

Parameter	Description
PubID	Specify the PubID of the message.
PubNode	Specify the Pub Node Name of the message.
ChannelName	Specify the channel name of the message.
VersionName	Specify the version name of the message.
MessageName	Specify the name of the message.

Returns

A reference to an XmlDoc object if successful, NULL if not successful.

See Also

Chapter 1, PeopleCode Built-in Functions, ReSubmitSubXmlDoc , page 376.

GetSyncLogData

Syntax

```
GetSyncLogData(GUID, pubnode, chnlname, msg_name, logtype [, message_version])
```

Description

Use the **GetSyncLogData** to return a log containing information about the specified synchronous message. You can use this information for debugging. Using this function, you can obtain the request and response data in a synchronous request, both pre- and post-transformation.

This function is used in the PeopleCode for the Message Monitor.

Parameters

Parameter	Description
GUID	Specify the GUID for the published synchronous message as a string. This property is populated after the message is sent.
pubnode	Specify the name of the node that the synchronous message was published from as a string.
chnlname	Specify the name of the channel the synchronous message was published to as a string.
msg_name	Specify the message definition name that you want to retrieve log data from as a string.
Log_type	Specify the type of log data you want to obtain, as a number. Values are: 1: the original request 2: the transformed request 3: the original response 4: the transformed response
message_version	Specify the message version name as a string.

Returns

An XML string containing the log data.

Example

```
Local String &guid, &pubnode, &channel, &msg_name;
Local Number &log_type;
..
..
&descrlong = GetSyncLogData(&guid, &pubnode, &channel, &msg_name, &log_type);
```

See Also

PeopleSoft Integration Broker, Using Integration Broker Monitor.

GetTreeNodeParent

Syntax

GetTreeNodeParent(*node*)

Description

Use the **GetTreeNodeParent** function to access data from dynamic tree controls.

Note. Dynamic tree controls have been deprecated. Use **GenerateTree** or Tree Viewer.

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateTree , page 230, Appendix B, Appendix B: Viewing Trees From Application Pages , page 1939.

GetTreeNodeRecordName

Syntax

```
GetTreeNodeRecordName(node)
```

Description

Use the **GetTreeNodeRecordName** function in accessing data from dynamic tree controls.

Note. Dynamic tree controls have been deprecated. Use **GenerateTree** or Tree Viewer.

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateTree , page 230, Appendix B, Appendix B: Viewing Trees From Application Pages , page 1939.

GetTreeNodeValue

Syntax

```
GetTreeNodeValue(node, [recordname.]fieldname)
```

Description

Use the **GetTreeNodeValue** function in accessing data from dynamic tree controls.

Note. Dynamic tree controls have been deprecated. Use **GenerateTree** or Tree Viewer.

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateTree , page 230, Appendix B, Appendix B: Viewing Trees From Application Pages , page 1939.

GetURL

Syntax

```
GetURL(URL.URLIdentifier)
```

Description

The **GetURL** function returns the URL, as a string, for the specified *URLIdentifier.* The *URLIdentifier* must exist and been created using URL Maintenance.

Note. If the URL identifier contains spaces, you must use quotation marks around *URLIdentifier*. For example, `GetURL(URL."My URL");`

Parameters

Parameter	Description
URLIdentifier	Specify a URL Identifier for a URL that already exists and was created using the URL Maintenance page.

Returns

A string containing the URL value for that URL Identifier, using the user's language preference.

Example

Suppose you have a URL with the identifier PEOPLESOFT, and the following URL:

```
http://www.peoplesoft.com
```

From the following code example

```
&PS_URL = GetURL(URL.PEOPLESOFT);
```

&PS_URL has the following value:

```
http://www.peoplesoft.com
```

Suppose you have the following URL stored in the URL Maintenance, with the name QE_CALL:

```
/S/WEBLIB_QE_MCD.QE_MCD_MAIN.FieldFormula.iScript_Call
```

You could combine this in the following code to produce an HTML string used as part of a response:

```
&output = GetHTMLText(HTML.QE_PHONELIST, %Request.RequestURI | "?" | ⇒
GetURL(URL.QE_CALL));
```

See Also

Chapter 1, PeopleCode Built-in Functions, ViewURL , page 498, *PeopleSoft Server Tools Administration*, PeopleTools Utilities, URL Maintenance.

GetUserOption

Syntax

GetUserOption(*Level*, *OPTN*)

Description

Use the **GetUserOption** function to return the default value for the specified option.

Parameters

Parameter	Description
Level	Specify the option category level as a string.
OPTN	Specify the option as a string.

Returns

The default value for the specified option.

Example

```
Local Any &MyValue;
```

```
&MyValue = GetUserOption("PPLT", "TZONE");
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetUserOption , page 437, *PeopleTools Security*, Managing PeopleSoft Personalizations.

GetWLFieldValue

Syntax

GetWLFieldValue(*fieldname*)

Description

When the user has opened a page from a Worklist (by selecting one of the work items) **GetWLFieldValue** retrieves the value of a field from the current row of the application Worklist record. You can use the **%WLName** system variable to check whether the page was accessed from a Worklist.

Returns

Returns the value of a specified field in the Worklist record as an Any data type.

Example

This example, from RowInit PeopleCode, populates page fields with values from the Worklist record. The %WLName system variable is used to determine whether there is a currently active Worklist (that is, whether the user accessed the page using a Worklist).

```
&WL = %WLName;
If &WL > " " Then
   &TEMP_NAME = "ORDER_NO";
   ORDER_NO = GetWLFieldValue(&TEMP_NAME);
   &TEMP_NAME = "BUSINESS_UNIT";
   BUSINESS_UNIT = GetWLFieldValue(&TEMP_NAME);
   &TEMP_NAME = "SCHED_Date";
   &SCHED_Date = GetWLFieldValue(&TEMP_NAME);
   SCHED_Date = &SCHED_Date;
   &TEMP_NAME = "DEMAND_STATUS";
   DEMAND_STATUS = GetWLFieldValue(&TEMP_NAME);
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, MarkWLItemWorked , page 318, Chapter 1, PeopleCode Built-in Functions, TriggerBusinessEvent , page 481, Chapter 3, System Variables , %WLName , page 581.

Global

Syntax

Global *data_type &var_name*

Description

The **Global** statement enables you to declare PeopleCode global variables. A global variable, once declared in any PeopleCode program, remains in scope throughout the PeopleSoft session. The variable must be declared with the **Global** statement in any PeopleCode program in which it is used.

Declarations tend to appear at the beginning of the program, intermixed with function declarations.

Not all PeopleCode data types can be declared as Global. For example, ApiObject data types can only be declared as Local.

Parameters

Parameter	Description
data_type	Specify a PeopleCode data type.
&var_name	A legal variable name.

Example

The following example declares a global variable and then assigns it the value of a field:

```
global string &AE_APPL_ID;
&AE_APPL_ID = AE_APPL_ID;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Local , page 315, Chapter 1, PeopleCode Built-in Functions, Component , page 74, *PeopleCode Developer's Guide*, PeopleCode Language, Data Types.

Gray

Syntax

Gray(*scrollpath*, *target_row*, [*recordname*.]*fieldname*)

where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD**.*target_recname*

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

If you put the function on the same scroll level as the field being changed, you can use the following syntax:

Gray(*Fieldname*)

The more complex syntax can be used to loop through a scroll on a lower level than the PeopleCode program.

Description

Gray makes a field unavailable for entry a page field, preventing the user from making changes to the field.

Note. This function remains for backward compatibility only. Use the **Enabled** Field class property instead.

Gray makes a field display-only, while **Hide** makes it invisible. You can undo these effects using the built-in functions **Ungray** and **Unhide**.

Note. If you specify a field as Display Only in PeopleSoft Application Designer, using the PeopleCode functions Gray, followed by Ungray, will not make it editable. This function shouldn't be used in any event prior to RowInit.

See Also

Chapter 12, Field Class , Enabled , page 855, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	An integer specifying the row on the target scroll level where the referenced buffer field is located.
[*recordname* **.**]*fieldname*	The name of the field to gray. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the call to **Gray** is in a record definition other than *recordname*.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Example

This example, which would typically be found in the RowInit event, disables the page's address fields if the value of the SAME_ADDRESS_EMPL field is "Y".

```
If SAME_ADDRESS_EMPL = "Y" Then
   Gray(STREET1);
   Gray(STREET2);
   Gray(CITY);
   Gray(STATE);
   Gray(ZIP);
   Gray(COUNTRY);
   Gray(HOME_PHONE);
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Hide , page 285, Chapter 1, PeopleCode Built-in Functions, Ungray , page 484, Chapter 1, PeopleCode Built-in Functions, Unhide , page 486.

GrayMenuItem

Syntax

GrayMenuItem(**BARNAME.***menubar_name*, **ITEMNAME.***menuitem_name*)

Description

Note. The **GrayMenuItem** function is supported for compatibility with previous releases of PeopleTools. New applications should use **DisableMenuItem** instead.

See Also

Chapter 1, PeopleCode Built-in Functions, DisableMenuItem , page 146.

Hash

Syntax

Hash(*ClearTextString*)

Description

The Hash function returns a fixed length value, based on the input. The output is always 28 characters. The input is variable length, with no maximum size. The same input *always* generates the same output. Different inputs may generate the same output, but the chances of this are rare. Because of this, hash output should not be used as a unique key to a table of data. Given the output of hash, it is impossible to determine the input.

Some of the original data is deliberately lost during the conversion process. This way, even if you know the algorithm, you can't "un-hash" the data.

Generally the Hash function is used like a checksum, to compare hashed values to ensure they match.

Parameters

Parameter	Description
ClearTextString	Specify the string you want converted.

Returns

A hash string.

Example

```
MessageBox("Please confirm password");

&HASHPW = Hash(&PASSWD);
&OPERPSWD = USERDEFN.OPERPSWD.Value;

If not (&HASHPW = &OPERPSWD) Then
   /* do error handling */
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Decrypt , page 133, Chapter 1, PeopleCode Built-in Functions, Encrypt , page 159.

HermiteCubic

Syntax

HermiteCubic(*DataPoints*)

Description

The **HermiteCubic** function computes a set of interpolating equations for a set of at least three datapoints. This particular Hermitian cubic is designed to mimic a hand-drawn curve.

Parameters

Parameter	Description
DataPoints	This parameter takes an array of array of number. The array's contents are an array of six numbers. The first two of these six numbers are the x and y points to be fit. The last four are the four coefficients to be returned from the function: **a**, **b**, **c** and **d**. **a** is the coefficient of the x^0 term, **b** is the coefficient of the x^1 term, **c** is the coefficient of the x^2 term, and **d** is the coefficient of the x^3 term.

Returns

A modified array of array of numbers. The elements in the array correspond to the elements in the array used for *DataPoints*.

See Also

Chapter 1, PeopleCode Built-in Functions, CubicSpline , page 112, Chapter 1, PeopleCode Built-in Functions, LinearInterp , page 314.

Hide

Syntax

Hide(*scrollpath*, *target_row*, [*recordname*.]*fieldname*)

where *scrollpath* is:

[**RECORD**.*level1_recname*, *level1_row*, [**RECORD**.*level2_recname*, *level2_row*,]] **RECORD**.*target_recname*

To prevent ambiguous references, you can also use **SCROLL**.*scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

The **Hide** function makes a page field invisible.

Note. This function remains for backward compatibility only. Use the **Visible** Field class property instead.

You can display the field again using **Unhide**, but **Unhide** has no effect on a field that has been made display-only in the page definition.

Gray, **Hide**, **Ungray**, and **Unhide** usually appear in RowInit programs that set up the initial display of data, and in FieldChange programs that change field display based on changes the user makes to a field. Generally, you put the functions on the same scroll level as the field that is being changed. This reduces the complexity of the function's syntax to:

```
Hide(fieldname)
```

The more complex syntax can be used to loop through a scroll on a lower level than the PeopleCode program.

Note. This function shouldn't be used in any event prior to RowInit.

See Also

Chapter 12, Field Class , Visible, page 870, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	An integer specifying the row on the target scroll level where the referenced buffer field is located.
[*recordname* **.**]*fieldname*	The name of the field to hide. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the call to Hide is in a record definition other than *recordname*.

Returns

Boolean (optional). Hide returns a Boolean value indicating whether it executed successfully.

Example

This example hides the page's address fields if SAME_ADDRESS_EMPL is equal to "Y":

```
If SAME_ADDRESS_EMPL = "Y" Then
   Hide(STREET1);
   Hide(STREET2);
   Hide(CITY);
   Hide(STATE);
   Hide(COUNTRY);
   Hide(HOME_PHONE);
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Gray , page 282, Chapter 1, PeopleCode Built-in Functions, Ungray , page 484, Chapter 1, PeopleCode Built-in Functions, Unhide , page 486.

HideMenuItem

Syntax

HideMenuItem(**BARNAME.***menubar_name*, **ITEMNAME.***menuitem_name*)

Description

HideMenuItem hides a specified menu item. To apply this function to a pop-up menu, use the PrePopup Event of the field with which the pop-up menu is associated.

If you're using this function with a pop-up menu associated with a page (not a field), the earliest event you can use is the **PrePopup** event for the first "real" field on the page (that is, the first field listed in the **Order** view of the page in PeopleSoft Application Designer.)

When a menu is first displayed, all menus are visible by default, so there is no need for a function to re-display a menuitem that has been hidden.

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Parameters

Parameter	Description
menubar_name	Name of the menu bar that owns the menuitem, or, in the case of pop-up menus, the name of the pop-up menu that owns the menuitem.
menuitem_name	Name of the menu item.

Returns

None.

Example

```
HideMenuItem(BARNAME.MYPOPUP1, ITEMNAME.DO_JOB_TRANSFER);
```

See Also

Chapter 1, PeopleCode Built-in Functions, DisableMenuItem , page 146, Chapter 1, PeopleCode Built-in Functions, EnableMenuItem , page 157, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, PrePopup Event.

HideRow

Syntax

HideRow(*scrollpath*) [, *target_row*])

Where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD**.*target_recname*

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

HideRow is used to hide a row occurrence programmatically.

Note. This function remains for backward compatibility only. Use the **Visible** Row class property instead.

It hides the specified row and any associated rows at lower scroll levels.

Hiding a row just makes the row invisible, it does not affect database processing such as inserting new rows, updating changed values, or deleting rows.

When you hide a row, it becomes the last row in the scroll or grid, and the other rows are renumbered accordingly. If you later use **UnHideRow** to make the row visible again, it is *not* moved back to its original position, but remains in its new position. When **HideRow** is used in a loop, you have to process rows from the highest number to the lowest to achieve the correct results.

Note. HideRow cannot be executed from the same scroll level as the row that is being hidden, or from a lower scroll level. Place the PeopleCode in a higher scroll level record.

See Also

Chapter 26, Row Class , Visible, page 1477, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	An integer specifying which row in the scroll to hide. If this parameter is omitted, the row on which the PeopleCode program is executing is assumed.

Returns

Boolean (optional). HideRow returns a Boolean value indicating whether the function executed successfully.

Example

This example hides all rows in scroll level 1 where the EXPORT_SW field is equal to "Y". Note that the loop has to count backwards from **ActiveRowCount** to 1.

```
For &ROW = ActiveRowCount(RECORD.EXPORT_OBJECT) to 1
step - 1
      &EXPORT_SW = FetchValue(EXPORT_OBJECT.EXPORT_SW, &ROW);
      If &EXPORT_SW  "Y" Then
         HideRow(RECORD.EXPORT_OBJECT, &ROW);
      Else
         /* WinMessage("not hiding row " | &ROW);*/
      End-if;
```

```
End-for;
```

See Also

Chapter 1, PeopleCode Built-in Functions, UnhideRow , page 487, Chapter 1, PeopleCode Built-in Functions, DeleteRow , page 140.

HideScroll

Syntax

HideScroll(*scrollpath*)

Where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD.***target_recname*

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

HideScroll is used to programmatically hide a scroll bar and all data items within the scroll.

Note. This function remains for backward compatibility only. Use the **HideAllRows** Rowset class method instead.

Typically this function is used in RowInit and FieldChange PeopleCode to modify the page based on user action.

See Also

Chapter 27, Rowset Class , HideAllRows, page 1493, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.

Returns

Boolean. **HideScroll** returns a Boolean value indicating whether the function executed successfully.

Example

This example, from RowInit PeopleCode, initializes the visibility of the scroll based on a field setting:

```
If %Component = COMPONENT.APPR_RULE Then
   If APPR_AMT_SW = "N" Then
      HideScroll(RECORD.APPR_RULE_LN, CurrentRowNumber(1), RECORD.APPR_RULE_DETL,⇒
 CurrentRowNumber(2), RECORD.APPR_RULE_AMT);
   Else
      UnhideScroll(RECORD.APPR_RULE_LN, CurrentRowNumber(1), ⇒
RECORD.APPR_RULE_DETL, CurrentRowNumber(2), RECORD.APPR_RULE_AMT);
```

```
   End-If;
End-If;
```

The corresponding FieldChange PeopleCode dynamically changes the appearance of the page based on user changes to the APPR_AMT_SW field:

```
If APPR_AMT_SW = "N" Then
   HideScroll(RECORD.APPR_RULEs_LN, CurrentRowNumber(1), RECORD.APPR_RULE_DETL,⇒
 CurrentRowNumber(2), RECORD.APPR_RULE_AMT);
   &AMT_ROWS = ActiveRowCount(RECORD.APPR_RULE_LN, CurrentRowNumber(1),⇒
 RECORD.APPR_RULE_DETL, CurrentRowNumber(2), RECORD.APPR_RULE_AMT);
   For &AMT_LOOP = &AMT_ROWS To 1 Step - 1
      DeleteRow(RECORD.APPR_RULE_LN, CurrentRowNumber(1), RECORD.APPR_RULE_DETL,⇒
 CurrentRowNumber(2), RECORD.APPR_RULE_AMT, &AMT_LOOP);
   End-For;
Else
   UnhideScroll(RECORD.APPR_RULE_LN, CurrentRowNumber(1), RECORD.APPR_RULE_DETL,⇒
 CurrentRowNumber(2), RECORD.APPR_RULE_AMT);
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, UnhideRow , page 487, Chapter 1, PeopleCode Built-in Functions, HideRow , page 287, Chapter 1, PeopleCode Built-in Functions, UnhideScroll , page 488.

HistVolatility

Syntax

HistVolatility(*Closing_Prices*, *Trading_Days*)

Description

The **HistVolatility** function computes the historical volatility of a market-traded instrument.

Parameters

Parameter	Description
Closing_Prices	An array of number. The elements in this array contain a vector of closing prices for the instrument.
Trading_Days	The number of trading days in a year.

Returns

A number.

See Also

Chapter 1, PeopleCode Built-in Functions, ConvertRate , page 86.

Hour

Syntax

Hour(*time_value*)

Description

Hour is used to extract a Number value for the hour of the day based on a Time or Datetime value. The value returned is a whole integer and is not rounded to the nearest hour.

Parameters

Parameter	Description
time_value	A Datetime or Time value.

Returns

Returns a Number equal to a whole integer value from 0 to 23 representing the hour of the day.

Example

If &TIMEOUT contains a Time value equal to 04:59:59 PM, the following example sets &TIMEOUT_HOUR to 16:

```
&TIMEOUT_HOUR = Hour(&TIMEOUT);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Minute , page 326, Chapter 1, PeopleCode Built-in Functions, Second , page 393.

Idiv

Syntax

Idiv(*x, divisor*)

Description

The **Idiv** function is an explicit integer division operation. It divides one number (*x*) by another (*divisor*).

Parameters

Parameter	Description
X	Specify an integer to be divided.
divisor	Specify the integer used to divide the first parameter.

Returns

An integer value.

Example

The following example sets &I1 to 1 and &I2 to -1:

```
&I1 = Idiv(3, 2);

&I2 = Idiv(17, 10);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Mod , page 327, Chapter 1, PeopleCode Built-in Functions, Int , page 298, Chapter 1, PeopleCode Built-in Functions, Integer , page 299, Chapter 1, PeopleCode Built-in Functions, Round , page 381, Chapter 1, PeopleCode Built-in Functions, Truncate , page 481, Chapter 1, PeopleCode Built-in Functions, Value , page 493.

If

Syntax

```
If condition Then
   [statement_list_1]
[Else
   [statement_list_2]]
End-If
```

Description

Use the **If** statement to execute statements conditionally, depending on the evaluation of a conditional expression. The **Then** and **Else** clauses of an **If** consist of arbitrary lists of statements. The **Else** clause may be omitted. If *condition* evaluates to True, all statements in the **Then** clause are executed; otherwise, all statements in the **Else** clause are executed.

Example

The following example's first **If** statement checks for BEGIN_DT and RETURN_DT, and makes sure that RETURN_DT is greater (later) than BEGIN_DT. If this is True, the execution continues at the following line, otherwise execution continues at the line beginning with **WinMessage**:

```
If All(BEGIN_DT, RETURN_DT) and
      BEGIN_DT = RETURN_DT Then
   &DURATION_DAYS = RETURN_DT - BEGIN_DT;
   If &DURATION_DAYS  999 Then
      DURATION_DAYS = 999;
   Else
      DURATION_DAYS = &DURATION_DAYS;
   End-if;
Else
   WinMessage("The beginning date is later then the return date!");
End-if;
```

InitChat

Syntax

InitChat(*logical queue ID*, *application data URL*, *customer username*, [*chat_subject*][, *chat_question*][, *wizard_URL*][, *priority*][, *skill_level*])

Description

InitChat enables a customer to engage an agent in a chat session. It places a chat request on a MultiChannel Framework queue and immediately launches a customer chat window. When an agent accepts this task from the queue, the system launches an agent chat window.

Note. Chats are implicitly queued with the current language setting of the initiator (%Language). Chats are only assigned to agents who have this language in their language list as specified on the Languages page of the Agents component.

Parameters

Parameter	Description
logical queue ID	Specifies the logical queue in which the task should be queued. It is a string value. The logical queue ID is a case-sensitive value. The case used in the **InitChat** function must exactly match the case used when creating the logical queue ID on the Queues page.
application data relative URL	This is the relative URL of the application page you want the agent to see on the left side of the agent-to-customer chat window when the agent accepts the chat. This value needs to be provided by your program. **Note.** This URL parameter must *not* point to content that includes JavaScript that breaks surrounding frames or that references the "top" window. In addition, the application page should not contain URL links to such content. This is because the Agent Chat Console is framed by the RenServer API which sends and receives the chat text.
customer username	This reflects the name of the customer or end user initiating the chat request. This value can be derived from the signon mechanism (%UserID) or by prompting the user. This is the name used to identify the chat requestor in the MultiChannel Framework chat console. For example, in the chat history window, all text sent by the customer is prefixed with this name.
chat_subject	This is an optional string parameter. The application can indicate a subject of the chat request. This could be prompted from the user or inferred from the page from which the chat is initiated. The system displays the subject on the agent's chat console when it assigns the chat to an agent.
chat_question	This is an optional string parameter. The application can indicate a specific question to be addressed in the chat. This could be prompted from the user or inferred from the page from which the chat is initiated. The value appears in the agent's chat window history box immediately after accepting the chat. This enables the agent to know the customer's question without having to ask.

Parameter	Description
wizard_URL	This feature leads the agent to an application page from which the agent can select a URL to push to the customer. This is an optional string parameter. This is the relative URL of the application page you wish the agent to launch when the agent clicks the Grab button on the Agent Chat console. If you do not provide this value, a default wizard is launched with no application-specific functionality. If you do provide this value, the application page must provide a wizard for pushing a URL to the customer. The wizard page provided by the application must be able to write the URL generated by the wizard to the URL field. The URL field is defined by PeopleSoft. In addition, you need to embed the HTML definition, MCF_GRABURL, which provides the Push and Push and Close buttons that push the URL in the URL field to the customer. See the Example section for examples showing the PeopleCode that would be used to generate the relative URL that is passed in to **InitChat** and the PeopleCode that would be used to embed the provided MCF_GRABURL definition into your application page.
priority	This is an optional parameter. It is an integer value expressing the priority level of the request. The minimum value is 0 and there is no maximum value. Specify the priority of this chat task. A higher value means a higher priority. MultiChannel Framework tasks are ordered on a physical queue based on their assigned priority, which means the system assigns a task of a higher priority before it assigns a task of a lower priority. If no value is specified, the system uses the default value specified for that task type on the Task Configuration page. When tasks have the same priority, the system orders the tasks according to time they were created. For example, suppose the following tasks exist: Priority 2 created at 11:15 AM and Priority 2 created at 11:16 AM. In this case, the system places the task created at 11:15 AM before the task created at 11:16 AM.
skill level	This is an optional parameter. It is an integer value expressing the minimum skill level required of the agent to whom the system routes the request. You set an agent's skill level in the Agent page. The minimum value is 0 and there is no maximum value. The queue server assigns this task type to an available agent on that queue with the lowest skill level greater than or equal to the skill level required by the task. If no value is specified, the system uses the default value specified for that task type in the Task Configuration page.

Returns

Returns a unique Chat ID in the form of an integer. You can use this ID to reference the chat in the chat log.

If unsuccessful, it returns a message number. The message set ID for MultiChannel Framework is 162.

For example, 1302 is returned when an invalid task type or no value is provided.

Example

For example, the following PeopleCode could be used to generate the relative URL that is passed in to **InitChat**.

```
&WizURL = GenerateComponentContentRelURL(%Portal, %Node, MenuName.PT_MCF, "GBL",⇒
 Component.MCF_DEMO_CMP, Page.MCF_URLWIZARD, "U");
```

The following is an example of embedding the provided MCF_GRABURL definition into your application page, using the **GetHTMLText** function.

```
Function IScript_GrabURL()
```

```
   &cssPTMCFDEF = %Response.GetStyleSheetURL(StyleSheet.PTMCFDEF);
   &cssPTSTYLEDEF = %Response.GetStyleSheetURL(StyleSheet.PTSTYLEDEF);
   &titleGrabURL = MsgGetText(162, 1170, "URL Wizard");
   &psDomain = SetDocDomainForPortal();
   If (&psDomain = "") Then
      &psDomain = SetDocDomainToAuthTokenDomain();
   End-If;
   &labelBtPush = MsgGetText(162, 1181, "Push");
   &labelBtPushClose = MsgGetText(162, 1186, "Push and Close");
   &HTML = GetHTMLText(HTML.MCF_GRABURL, &titleGrabURL, &cssPTSTYLEDEF, ⇒
&cssPTMCFDEF, &psDomain, &labelBtPush, &labelBtPushClose);
   %Response.Write(&HTML);
End-Function;
```

The following is an example of the usage of **InitChat.**

```
&ret = InitChat("SALES", "http://www.support.company.com/products.html",
"John Smith", "Widgets", "How to order widgets", "", 2, 2);
```

The following example illustrates how to pass a PeopleCode-generated URL using the GenerateComponentContentRelURL function for Application Data URL and wizards.

```
    &urlTestComponent = GenerateComponentContentRelURL(%Portal, %Node, Menu⇒
Name.UTILITIES, "GBL", Component.MESSAGE_CATALOG1, Page.MESSAGE_CATALOG, "U");

   &WizURL = GenerateComponentContentRelURL(%Portal, %Node, MenuName.PT_MCF,⇒
 "GBL", Component.MCF_DEMO_CMP, Page.MCF_URLWIZARD, "U");

   try

      &ret = InitChat(&QUEUEID, &urlTestComponent, &Username, &subject, &question,⇒
 &wizurl, &priority, &minskill);
   catch Exception &E

      MessageBox(0, "", 0, 0, "Caught exception: " | &E.ToString());
   end-try;
```

InsertImage

Syntax

InsertImage(*scrollpath*, *target_row*, [*recordname*.]*fieldname*)

where *scrollpath* is:
[**SCROLL.***level1_recname*, *level1_row*, [**SCROLL.***level2_recname*, *level2_row*,]]
SCROLL.*target_recname*

Description

The **InsertImage** function enables a user to associate an image file with a record field on a page. After the image file is associated with the record field, it can be saved to the database when the component is saved.

The following are the valid types of image files that can be associated with a record field:

- JPEG
- BMP
- DIB

InsertImage uses a search page to enable the end-user to select the image file to be used. This is the same search page used to add an attachment.

Note. To update an image field using this function, be sure that PSIMAGEVER field is also on the same record as the image field being updated.

Restrictions on Use in PeopleCode Events

InsertImage is a "think-time" function, which means it shouldn't be used in any of the following PeopleCode events:

- SavePreChange
- SavePostChange
- Workflow
- RowSelect
- Any PeopleCode event that fires as a result of a ScrollSelect (or one of its relatives) function calls, or a Select (or one of its relatives) Rowset class method.

See *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Think-Time Functions.

Image Size Considerations

The size of the image that can be saved to the database depends on the database platform.

Database Platform	Size limitation
DB2/400	30 KB
DB2/MVS	31 KB
DB2/Unix	31 KB

Platforms other than those listed here are effectively without a size limit, that is, they allow for images equal to or greater than 2 GB.

Platform size limitations are subject to change.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll area in the component buffer.
target_row	The row number of the target row.
[*recordname* **.**]*fieldname*	The name of the field to be associated with the image file. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the function call is in a record definition other than *recordname*.

Returns

The InsertImage function returns either a constant or a number:

Numeric Value	Constant Value	Description
0	%InsertImage_Success	Image was successfully associated with the record field.
1	%InsertImage_Failed	Image was not successfully associated with the record field. When the component is saved the image file will not be saved to the database.
2	%InsertImage_Canceled	User canceled the transaction so image file isn't associated with record field.
3	%InsertImage_ExceedsMaxSize	

Example

```
&RC = InsertImage(EMPL_PHOTO.EMPLOYEE_PHOTO);
```

See Also

Chapter 1, PeopleCode Built-in Functions, DeleteImage , page 138.

InsertRow

Syntax

```
InsertRow(scrollpath, target_row [, turbo])
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row,
] RECORD.target_recname
```

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

Use the **InsertRow** function to programmatically perform the ALT+7 and ENTER (**RowInsert**) function.

Note. This function remains for backward compatibility only. Use the **InsertRow** Rowset method instead.

InsertRow inserts a new row in the scroll buffer and causes a RowInsert PeopleCode event to fire, followed by the events that normally follow a RowInsert, as if the user had manually pressed ALT+7 and ENTER.

InsertRow can be executed in turbo or non-turbo mode. To execute the function in turbo mode, pass a value of True in the optional *turbo* parameter. Non-turbo mode is the default. In turbo mode, default processing is performed only for the row being inserted.

In scrolls that are not effective-dated, the new row is inserted *after* the target row specified in the function call. However, if the scroll is effective-dated, then the new row is inserted *before* the target row, and all the values from the previous current row are copied into the new row, except for EffDt, with is set to the current date.

Note. InsertRow cannot be executed from the same scroll level where the insertion will take place, or from a lower scroll level. Place the PeopleCode in a higher scroll level record.

See Also

Chapter 27, Rowset Class , InsertRow, page 1494, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number indicating the position where the new row will be inserted.
turbo	Specifies whether default processing is performed on the entire table or just the row being inserted. Pass a value of True to execute this parameter. Non-turbo mode is the default.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

The example inserts a row on the level-two page scroll. The PeopleCode has to be in the scroll-level-one record:

```
InsertRow(RECORD.BUS_EXPENSE_PER, &L1_ROW, RECORD.BUS_EXPENSE_DTL, &L2_ROW);
```

See Also

Chapter 1, PeopleCode Built-in Functions, DeleteRow , page 140, Chapter 1, PeopleCode Built-in Functions, HideRow , page 287, Chapter 1, PeopleCode Built-in Functions, UnhideRow , page 487.

Int

Syntax

Int(*decimal*)

Description

Int truncates a decimal number *x* to an integer and returns the result as a Number value.

Note. PeopleSoft only supports 32 bit integers. The largest integer value we support is 4294967295.

Parameters

Parameter	Description
decimal	A decimal number to be truncated.

Returns

Returns a Number equal to *decimal* truncated to a whole integer.

Example

The following example sets &I1 to 1 and &I2 to -4:

```
&I1 = Int(1.975);
&I2 = Int(-4.0001);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Mod , page 327, Chapter 1, PeopleCode Built-in Functions, Round , page 381, Chapter 1, PeopleCode Built-in Functions, Truncate , page 481, Chapter 1, PeopleCode Built-in Functions, Value , page 493.

Integer

Syntax

Integer(*decimal*)

Description

The **Integer** function converts *decimal* to an integer (32 bit signed twos complement number) by truncating any fraction part towards zero and returns the result as an Integer value.

Differences between Int and Integer

There is one primary difference between the **Int** function and the **Integer** function.

- The **Int** function rounds to a number in floating-decimal-point representation with a range of -9,999,999,999,999,999,999,999,999,999,999,999 to 9,999,999,999,999,999,999,999,999,999,999,999.
- The **Integer** function truncates to a number in 32 bit binary twos-complement representation with a range of -2,147,483,648 to 2,147,483,647.

Parameters

Parameter	Description
decimal	A decimal number to be truncated to an integer.

Returns

Returns an Integer equal to *decimal* truncated to a whole integer. If *decimal* is outside the range that can be represented by an integer type, the result isn't defined.

See Also

Chapter 1, PeopleCode Built-in Functions, Mod , page 327, Chapter 1, PeopleCode Built-in Functions, Int , page 298, Chapter 1, PeopleCode Built-in Functions, Round , page 381, Chapter 1, PeopleCode Built-in Functions, Truncate , page 481, Chapter 1, PeopleCode Built-in Functions, Value , page 493.

IsAlpha

Syntax

IsAlpha(*String*)

Description

Use the **IsAlpha** function to determine if *String* contains only alphabetic characters. This function works for both Unicode as well as non-Unicode strings. Alphabetic characters include accented characters.

Parameters

Parameter	Description
String	Specify the string you want to search for alphabetic characters.

Returns

A Boolean value: True if the string contains only alphabetic characters, False otherwise.

Example

```
&Value = Get Field().Value;
If IsAlpha(&Value) Then
   /* do alphabetic processing */
Else
   /* do non-alphabetic processing */
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, IsAlphaNumeric , page 300, Chapter 1, PeopleCode Built-in Functions, IsDigits , page 304, Chapter 1, PeopleCode Built-in Functions, IsDate , page 301, Chapter 1, PeopleCode Built-in Functions, IsDateTime , page 302, Chapter 1, PeopleCode Built-in Functions, IsNumber , page 309, Chapter 1, PeopleCode Built-in Functions, IsTime , page 311.

IsAlphaNumeric

Syntax

IsAlphaNumeric(*String*)

Description

Use the **IsAlphaNumeric** function to determine if *String* contains only alphanumeric characters. This function works for both Unicode as well as non-Unicode strings.

Alphabetic characters include accented characters. Numeric characters include sign indicators and comma and period decimal points.

Parameters

Parameter	Description
String	Specify the string you want to search for alphanumeric characters.

Returns

A Boolean value: True if the string contains only alphanumeric characters, False otherwise.

Example

```
&Value = Get Field().Value;
If IsAlphaNumeric(&Value) Then
   /* do alphanumeric processing */
Else
   /* do non-alphanumeric processing */
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, IsNumber , page 309, Chapter 1, PeopleCode Built-in Functions, IsAlpha , page 300.

IsDate

Syntax

IsDate(*Value*)

Description

Use the **IsDate** function to determine if *Value* contains a valid date.

You can use this function when you want to determine if a value is compatible with the **Date** built-in function.

Parameters

Parameter	Description
Value	Specify either a string or number you want to search for a valid date. *Value* is a number or string of the format *yyyymmdd*.

Returns

A Boolean value: True if the string contains a valid date, False otherwise.

Example

```
If IsDate(&Num) Then
   &Datevalue = Date(&Num);
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, IsAlphaNumeric , page 300, Chapter 1, PeopleCode Built-in Functions, IsDigits , page 304, Chapter 1, PeopleCode Built-in Functions, IsDateTime , page 302, Chapter 1, PeopleCode Built-in Functions, IsNumber , page 309, Chapter 1, PeopleCode Built-in Functions, IsTime , page 311.

IsDateTime

Syntax

```
IsDateTime(String)
```

Description

Use the **IsDateTime** function to determine if *String* contains a datetime string in the standard PeopleSoft format, that is, in the following format:

```
yyyy-mm-dd hh:mm:ss.mmmmmm
```

Parameters

Parameter	Description
String	Specify the string you want to search for a valid datetime.

Returns

A Boolean value: True if the string contains a valid datetime, False otherwise.

Example

The following example uses the short form of dot notation, by combining the getting the field value with getting the value of IsDateTime and making it a conditional statement:

```
If IsDateTime(GetField().Value) Then
   /* do date processing */
Else
   /* do non-date processing */
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, IsAlphaNumeric , page 300, Chapter 1, PeopleCode Built-in Functions, IsDigits , page 304, Chapter 1, PeopleCode Built-in Functions, IsDate , page 301, Chapter 1, PeopleCode Built-in Functions, IsNumber , page 309, Chapter 1, PeopleCode Built-in Functions, IsTime , page 311.

IsDaylightSavings

Syntax

```
IsDaylightSavings(datetime, {timezone | "Local" | "Base"});
```

Description

IsDaylightSavings returns a Boolean value based on whether daylight savings is active in the specified time zone at the specified datetime. For time zones that don't observe daylight savings time, this function always returns False.

The system's base time zone is specified on the PSOPTIONS table.

Parameters

Parameter	Description
datetime	The datetime value you want to check.
timezone | **Local** | **Base**	Specify a value for converting *datetime*. The values are: • *timezone* - a time zone abbreviation or a field reference to be used for converting *textdatetime*. • **Local** - use the local time zone for converting *textdatetime*. • **Base** - use the base time zone for converting *textdatetime*.

Returns

A Boolean value: True if daylight savings is active in the specified time zone at the specified datetime. Returns False otherwise.

Example

In the first example, TESTDTTM has value of 01/01/99 10:00:00AM. &OUTPUT is False.

```
&OUTPUT = IsDaylightSavings(TESTDTTM, "EST")
```

In this example, TESTDTTM has value of 04/05/99 12:00:00AM. &OUTPUT has a value of True: 12:00am PST = 3:00am EST, so Daylight Savings Time has switched on.

```
&OUTPUT = IsDaylightSavings(TESTDTTM, "EST")
```

In this example, TESTDTTM has value of 04/05/99 12:00:00AM. &OUTPUT returns False: 12:00am PST = 1:00am MST, so Daylight Savings Time hasn't started yet.

```
&OUTPUT = IsDaylightSavings(TESTDTTM, "MST")
```

In this example, TESTDTTM has value of 07/07/99 10:00:00. &OUTPUT returns False: ESTA is Indiana time, where they do not observe Daily Savings Time.

```
&OUTPUT = IsDaylightSavings(TESTDTTM, "ESTA")
```

See Also

Chapter 1, PeopleCode Built-in Functions, ConvertDatetimeToBase , page 85, Chapter 1, PeopleCode Built-in Functions, ConvertTimeToBase , page 87, Chapter 1, PeopleCode Built-in Functions, DateTimeToTimeZone , page 124, Chapter 1, PeopleCode Built-in Functions, TimeToTimeZone , page 460, Chapter 1, PeopleCode Built-in Functions, TimeZoneOffset , page 461, *PeopleSoft Server Tools Administration*, PeopleTools Utilities.

IsDigits

Syntax

IsDigits(*String*)

Description

Use the **IsDigits** function to determine if *String* contains only digit (numeric) characters. This function works for both Unicode as well as non-Unicode strings. Numeric characters do *not* include sign indicators and comma and period decimal points. If you want to check for numbers as well as sign indicators, use the **IsNumber** function.

Parameters

Parameter	Description
String	Specify the string you want to search.

Returns

A Boolean value: True if the string contains digits, False otherwise.

Example

```
If IsDigits(&MyValue) Then
   /* do processing */
Else
   /* do error processing */
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, IsAlpha , page 300, Chapter 1, PeopleCode Built-in Functions, IsAlphaNumeric , page 300, Chapter 1, PeopleCode Built-in Functions, IsDate , page 301, Chapter 1, PeopleCode Built-in Functions, IsDateTime , page 302, Chapter 1, PeopleCode Built-in Functions, IsNumber , page 309, Chapter 1, PeopleCode Built-in Functions, IsTime , page 311.

IsHidden

Syntax

IsHidden(*scrollpath*, *target_row*)

Where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,] **RECORD**.*target_recname*

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

IsHidden checks to see whether a row is hidden or not.

Note. This function remains for backward compatibility only. Use the **Visible** Row class property instead.

It returns True if the row is hidden, otherwise it returns False. **IsHidden** must be called in a PeopleCode program on a higher scroll level than one you are checking.

See Also

Chapter 26, Row Class , Visible, page 1477, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number indicating the position of the row.

Example

The following example tests whether a specific row on scroll level one is hidden:

```
&ROW_CNT = ActiveRowCount(RECORD.LD_SHP_INV_VW);
&FOUND = True;
If &ROW_CNT = 1 Then
   &ORDER = FetchValue(LD_SHP_INV_VW.ORDER_NO, 1);
   If None(&ORDER) Then
      &FOUND = False;
   End-If;
End-If;
If &FOUND Then
   For &I = 1 To &ROW_CNT
      If Not IsHidden(RECORD.LD_SHP_INV_VW, &I) Then
         UpdateValue(ITEM_SELECTED, &I, "N");
      End-If;
   End-For;
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, HideRow , page 287, Chapter 1, PeopleCode Built-in Functions, UnhideRow , page 487.

IsMenuItemAuthorized

Syntax

IsMenuItemAuthorized(**MENUNAME.***menuname*, **BARNAME.***barname*, **ITEMNAME.***menuitem_name*, **PAGE.***component_item_name*[, *action*])

Description

The **IsMenuItemAuthorized** function returns True if the current user is allowed to access the specified menu item.

Note. You don't need to use this function to gray internal link pushbuttons/hyperlinks. This function is generally used for transfers that are part of some PeopleCode processing.

Parameters

Parameter	Description
menuname	The name of the menu where the page is located, prefixed with the reserved word **MENUNAME**.
barname	The name of the menu bar where the page is located, prefixed with the reserved word **BARNAME**.
menu_itemname	The name of the menu item where the page is located, prefixed with the reserved word **ITEMNAME**.
component_item_name	Specify the component item name of the page to be displayed on top of the component when it displays. The component item name is specified in the component definition. If you specify a page, it must be prefixed with the keyword **PAGE**. You can also specify a null ("") for this parameter.
action	String representing the action mode in which to start up the page. If *action* is omitted, the current action is used. Values are:

Constant	Description
%Action_Add	Add
%Action_UpdateDisplay	Update/Display
%Action_UpdateDisplayAll	Update/Display All
%Action_Correction	Correction
%Action_DataEntry	Data Entry
%Action_Prompt	Prompt

Returns

A Boolean value: True if the user is authorized to access the specified page, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, DoModal , page 148, Chapter 1, PeopleCode Built-in Functions, DoModalComponent , page 150, Chapter 1, PeopleCode Built-in Functions, Transfer , page 463, Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475.

IsMessageActive

Syntax

IsMessageActive(**Message.***Message_Name*)

Description

Use the **IsMessageActive** built-in function to determine whether the specified message definition has been set to inactive in PeopleSoft Application Designer.

Note. This function is used only with XmlDoc messages, that is, after you get an XmlDoc message (using GetMessageXmlDoc) use this function to determine if the message is active. Use the IsActive message class property to determine if a structured message definition is active.

This function returns True if the message definition for the XmlDoc message is active, False if it's been inactivated from PeopleSoft Application Designer. If you have a lot of PeopleCode associated with publishing an XmlDoc message, you might use this function to check if the message is active before you publish it.

Parameters

Parameter	Description
Message_Name	Specify the name of the message you want to inquire about. The message name must be prefixed with the reserved work **Message.** This function is used only with XmlDoc message definitions.

Returns

A Boolean value: True if the message is active, False otherwise.

Example

```
&Active = IsMessageActive(Message.MyMessage);

If &Active Then
   /* code for getting message, populating it and publishing it */
End-if;
```

See Also

Chapter 17, Message Class , IsActive, page 1001, Chapter 34, XmlDoc Class , page 1757.

IsModal

Syntax

IsModal()

Description

IsModal returns True if executed from PeopleCode running in a modal secondary page and False if executed elsewhere. This function is useful in separating secondary page-specific logic from general PeopleCode logic.

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Parameters

None.

Returns

A Boolean value.

Example

The following example executes logic specific to a secondary page:

```
If Not IsModal() Or
      Not (%Page = PAGE.PAY_OL_REV_RUNCTL Or
         %Page = PAGE.PAY_OL_RE_ASSGN_C Or
         %Page = PAGE.PAY_OL_RE_ASSGN_S) Then
   Evaluate COUNTRY
   When = "USA"
   When = "CAN"
      If Not AllOrNone(ADDRESS1, CITY, STATE) Then
         Warning MsgGet(1000, 5, "Address should consist of at least Street (Line⇒
 1), City, State, and Country.")
      End-If;
      Break;
   When-Other;
      If Not AllOrNone(ADDRESS1, CITY, COUNTRY) Then
         Warning MsgGet(1000, 6, "Address should consist of at least Street (Line⇒
 1), City, and Country.")
      End-If;
   End-Evaluate;
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, DoModal , page 148, Chapter 1, PeopleCode Built-in Functions, EndModal , page 161.

IsModalComponent

Syntax

IsModalComponent()

Description

IsModalComponent tests whether a modal component is currently executing, enabling you to write PeopleCode that only executes when a component has been called with DoModalComponent.

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Parameters

None.

Returns

Returns a Boolean value: True if the current program is executing from a modal component, False otherwise.

Example

```
If IsModalComponent() then
/* Logic that executes only if component is executing modally.  */
end-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, DoModalComponent , page 150.

IsModalPanelGroup

Syntax

IsModalPanelGroup()

Description

IsModalPanelGroup tests whether a modal component is currently executing.

Note. The **IsModalPanelGroup** function is supported for compatibility with previous releases of PeopleTools. New applications should use **IsModalComponent** instead.

See Also

Chapter 1, PeopleCode Built-in Functions, IsModalComponent , page 308.

IsNumber

Syntax

IsNumber(*Value*)

Description

Use the **IsNumber** function to determine if *Value* contains a valid numeric value. Numeric characters include sign indicators and comma and period decimal points.

Parameters

Parameter	Description
Value	Specify a string you want to search to determine if it is a valid number.

Returns

A Boolean value: True if *Value* contains a valid numeric value, False otherwise.

Example

```
&Value = Get Field().Value;
If IsNumber(&Value) Then
   /* do numeric processing */
Else
   /* do non-numeric processing */
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, IsAlpha , page 300, Chapter 1, PeopleCode Built-in Functions, IsAlphaNumeric , page 300, Chapter 1, PeopleCode Built-in Functions, IsDigits , page 304, Chapter 1, PeopleCode Built-in Functions, IsDate , page 301, Chapter 1, PeopleCode Built-in Functions, IsDateTime , page 302, Chapter 1, PeopleCode Built-in Functions, IsTime , page 311.

IsOperatorInClass

Syntax

IsOperatorInClass(*operclass1* [*, operclass2*]. . .)

Description

IsInOperatorClass takes an arbitrary-length list of strings representing the names of operator classes and determines whether the current operator belongs to any class in a list of classes.

Note. The **IsOperatorInClass** function is supported for compatibility with previous releases of PeopleTools. New applications should use **IsUserInPermissionList** instead.

See Also

Chapter 1, PeopleCode Built-in Functions, IsUserInPermissionList , page 312.

IsSearchDialog

Syntax

IsSearchDialog()

Description

IsSearchDialog determines whether a search dialog, add dialog, or data entry dialog box is currently executing. Use it to make processes conditional on whether a search dialog box is running.

Returns

Returns a Boolean value: True if a search dialog box is executing, False otherwise.

Example

```
If Not (IsSearchDialog()) Then
   If %Component = COMPONENT.SALARY_GRADE_TBL Then
      If All(SALARY_MATRIX_CD) Then
         Gray(RATING_SCALE)
      End-If;
      calc_range_spread();
   End-If;
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetSearchDialogBehavior , page 429.

IsTime

Syntax

IsTime(*Value*)

Description

Use the **IsTime** function to determine if *Value* contains a valid Time.

You can use this function when you want to determine if a value is compatible with the **Time** built-in function.

Parameters

Parameter	Description
Value	Specify either a string or number you want to search for to determine if it's a valid Time.

Returns

A Boolean value: True if the string contains a valid Time value, False otherwise.

Example

```
If IsTime(&Num) Then
   &Timevalue = Time(&Num);
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, IsAlpha , page 300, Chapter 1, PeopleCode Built-in Functions, IsAlphaNumeric , page 300, Chapter 1, PeopleCode Built-in Functions, IsDigits , page 304, Chapter 1, PeopleCode Built-in Functions, IsDate , page 301, Chapter 1, PeopleCode Built-in Functions, IsDateTime , page 302, Chapter 1, PeopleCode Built-in Functions, IsNumber , page 309, Chapter 1, PeopleCode Built-in Functions, IsTime , page 311.

IsUserInPermissionList

Syntax

IsUserInPermissionList(*PermissionList1* [*, PermissionList2*]. . .)

Description

IsUserInPermissionList takes an arbitrary-length list of strings representing the names of Permission Lists and determines whether the current user belongs to any of the Permission Lists.

Parameters

Parameter	Description
PermissionList	An arbitrary-length list of string, each of which represents a Permission List.

Returns

Returns a Boolean value: True if the current user has access to one or more of the Permission Lists, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, ExecuteRolePeopleCode , page 173, Chapter 1, PeopleCode Built-in Functions, ExecuteRoleQuery , page 174, Chapter 1, PeopleCode Built-in Functions, ExecuteRoleWorkflowQuery , page 175, Chapter 1, PeopleCode Built-in Functions, IsUserInRole , page 312, Chapter 3, System Variables , %PermissionLists , page 575, Chapter 3, System Variables , %PrimaryPermissionList , page 575, Chapter 3, System Variables , %Roles , page 577, Chapter 3, System Variables , %RowSecurityPermissionList , page 577, *PeopleTools Security*, Working With Permission Lists.

IsUserInRole

Syntax

IsUserInRole(*rolename1* [*, rolename2*]. . .)

Description

IsUserInRole takes an arbitrary-length list of strings representing the names of roles and determines whether the current user belongs to any role in an array of roles.

Parameters

Parameter	Description
rolename	An arbitrary-length list of strings, each of which represents a role.

Returns

Returns a Boolean value: True if the current user belongs to one or more of the roles in the user role array, False otherwise.

See Also

PeopleTools Security, Working With Roles.

Left

Syntax

Left(*source_str, num_chars*)

Description

Left returns a substring containing the leftmost number of characters in *source_str*. *num_chars* specifies how many characters to extract.

Parameters

Parameter	Description
source_str	A String from which to derive the substring.
num_chars	A Number specifying how many characters to take from the left of *source_str*. The value of *num_chars* must be greater than or equal to zero. If *num_chars* is greater than the length of *source_str*, **Left** returns the entire string. If *num_chars* is omitted, it is assumed to be one.

Returns

Returns a String value derived from *source_str*.

Example

The following example sets &SHORT_ZIP to "90210":

```
&SHORT_ZIP = Left("90210-4455", 5);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Right , page 381.

Len

Syntax

Len(*str*)

Description

Use the **Len** function to determine the number of characters in a string.

Returns

Returns a Number value equal to the number of characters, including spaces, in *str*.

Example

The following example sets &STRLEN to 10, then to 0:

```
&STRLEN = Len("PeopleSoft");
&STRLEN = Len("");
```

See Also
Chapter 1, PeopleCode Built-in Functions, Lenb , page 314.

Lenb

Syntax
Lenb(*str*)

Description

Lenb determines the number of bytes in a string. This is useful to determine the size in bytes of a DBCS string.

Returns

Returns a Number value equal to the number of bytes in *str*.

Example

The following example sets &STRLEN to 20, if NAME is a field containing a string consisting of 10 double-byte characters:

```
&STRLEN = Lenb(NAME);
```

See Also
Chapter 1, PeopleCode Built-in Functions, Len , page 313.

LinearInterp

Syntax
LinearInterp(*DataPoints*)

Description

The **LinearInterp** function computes a set of lines through a sequence of at least two points.

Parameters

Parameter	Description
DataPoints	This parameter takes an array of array of number. The array's contents are an array of six numbers. The first two of these six numbers are the x and y points to be fit. The last four are the four coefficients to be returned from the function: **a**, **b**, **c** and **d**. **a** is the coefficient of the x^0 term, **b** is the coefficient of the x^1 term, **c** is the coefficient of the x^2 term, and **d** is the coefficient of the x^3 term.

Returns

A modified array of array of numbers. The elements in the array correspond to the elements in the array used for *DataPoints*. The **c** and **d** elements contain zeros.

See Also
Chapter 1, PeopleCode Built-in Functions, CubicSpline , page 112, Chapter 1, PeopleCode Built-in Functions, HermiteCubic , page 285.

Ln

Syntax

Ln(*i*)

Description

The **Ln** function determines the natural logarithm of a number. Natural logarithms are based on the constant e, which equals 2.71828182845904. The number *i* must be a positive real number. **Ln** is the inverse of **Exp**.

Returns

A Number equal to the natural logarithm of *i*.

Example

The following examples set &I to 2.302585 and &J to 1:

```
&I = Ln(10);
&J = Ln(2.7182818);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Exp , page 176, Chapter 1, PeopleCode Built-in Functions, Log10 , page 316.

Local

Syntax

Local *data_type* &*var_name* [= *expression*]

Description

Use the **Local** statement to explicitly define local variables in PeopleCode.

Variable declarations can appear at the start of the program, intermixed with function declarations. Local variable declarations can also appear in the body of a program, including inside functions.

The scope of local variables declared outside of a function is the PeopleCode program. The scope of local variables declared inside a function is to the end of the function. Also, these function local variables are different variables for each call of the function (even for recursive calls,) in the same manner as parameters.

Local variable declarations intermixed with the body of the program or inside functions can include initialization, that is, the variable can be set to a value at the same time it is declared.

In the absence of an initialization, the system automatically initializes temporary variables. Declared variables always have values appropriate to their declared type. Undeclared local variables are initialized as null strings.

Parameters

Parameter	Description
data_type	Any PeopleCode data type.
&*var_name*	A legal variable name.
expression	Specify the value of the variable. This parameter is optional.

Example

```
local string &LOC_FIRST;
```

The following example scopes the local variable &Constant as a local variable, as well as initializing it.

```
Local Number &Constant = 42;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Global , page 281, Chapter 1, PeopleCode Built-in Functions, Component , page 74, *PeopleCode Developer's Guide*, PeopleCode Language, Data Types.

Log10

Syntax

Log10(*x*)

Description

Log10 returns the base-10 logarithm of a number *x* as a Number value. The number *x* must be a positive real number.

Returns

Returns a Number equal to the base-10 logarithm of *x*.

Example

The following example sets &X to 1 and &Y to 1.39794:

```
&X = Log10(10);
&Y = Log10(25);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Exp , page 176, Chapter 1, PeopleCode Built-in Functions, Ln , page 315.

Lower

Syntax

Lower(*string*)

Description

Lower converts all uppercase characters in a text string to lowercase characters and returns the result as a String value. **Lower** does not change characters that are not letters. The lowercase conversion is based on the Windows language you have active. Refer to the Windows Control Panel to specify the language.

Returns

A String value equal to string, but in all lowercase format.

Example

The example sets &GOODKD to "k d lang":

```
&GOODKD = Lower("K D Lang");
```

See Also

Chapter 1, PeopleCode Built-in Functions, Proper , page 352, Chapter 1, PeopleCode Built-in Functions, Upper , page 492.

LTrim

Syntax

LTrim(*string1* [,*string2*])

Description

LTrim returns a string formed by deleting from the beginning of *string1*, all occurrences of each character in *string2*. If *string2* is omitted, " " is assumed; that is, leading blanks are trimmed.

Example

The following removes leading blanks from &NAME:

```
&TRIMMED = LTrim(&NAME);
```

The following removes leading punctuation marks from REC.INP:

```
&TRIMMED = LTrim(REC.INP, ".,;:!?");
```

See Also

Chapter 1, PeopleCode Built-in Functions, RTrim , page 387.

MarkPrimaryEmailAddress

Syntax

MarkPrimaryEmailAddress(*Type*)

Description

Use the **MarkPrimaryEmailAddress** function to specify which email address is the primary email address for the current user. You can only have one primary email address per user.

Parameters

Parameter	Description
Type	Specify the type that you want to change the email address to. This parameter takes a string value. The valid values are:

Value	Description
BB	Blackberry email address
BUS	Business email address
HOME	Home email address
OTH	Other email address
WORK	Work email address

Returns

None.

See Also

Chapter 1, PeopleCode Built-in Functions, AddEmailAddress , page 35, Chapter 1, PeopleCode Built-in Functions, ChangeEmailAddress , page 63, Chapter 1, PeopleCode Built-in Functions, DeleteEmailAddress , page 137.

MarkWLItemWorked

Syntax

```
MarkWLItemWorked()
```

Description

If you've invoked a page from the Worklist, you can mark the current Worklist entry as worked using this function. It should be called only in Workflow PeopleCode. You can use the %WLName system variable to check whether the page has been accessed using a Worklist.

Returns

Returns a Boolean value indicating whether it executed successfully. The return value is not optional.

Example

This example, which would be in the WorkFlow event, checks to see whether a page check box MARK_WORKED_SW is selected, and if so, it marks the item in the worklist as complete:

```
If MARK_WORKED_SW = "Y" Then
   If MarkWLItemWorked() Then
   End-If;
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetWLFieldValue , page 281, Chapter 3, System Variables , %WLName , page 581.

Max

Syntax

Max(*param_list*)

Where *param_list* has the form

parameter1, *parameter2* [, *parameter3*, . . . *parameterN*]

Description

Use the **Max** function to determine the maximum value in the parameter list. The type of every item in the parameter list must be compatible with the first parameter in the list.

For example, if the first parameter is a string and the second parameter is a number with value 123.456, the second parameter is converted to the string "123.456" before the comparison is performed.

If all the values in the parameter list are alpha characters, "Z" is greater than "A" so Max("Z", "A") returns "Z".

Parameters

Parameter	Description
param_list	Specify a list of items to be compared. All items in the parameter list must be of the same type. If a value isn't defined, the system assumes it's of the same type as the first parameter.

Returns

The item in the list that has the maximum value.

Example

```
&RSULT = Max("A", "B", "C", "D", "E");
```

See Also

Chapter 1, PeopleCode Built-in Functions, Min , page 326.

MessageBox

Syntax

MessageBox(*style*, *title*, *message_set*, *message_num*, *default_txt* [, *paramlist*])

where *paramlist* is an arbitrary-length list of parameters of undetermined (Any) data type to be substituted in the resulting text string, in the form:

param1 [, *param2*]. . .

Description

MessageBox displays a message box window. This function combines dialog-display ability with the text-selection functionality of **MsgGet**, **MsgGetText**, or **MsgGetExplainText**. The *style* parameter selects the buttons to be included. *title* determines the title of message.

Note. *title* isn't available for messages displayed in the PeopleSoft Internet Architecture. Also, *style* is ignored if the message has any severity other than Message.

The remaining parameters are used to retrieve and process a text message selected from the Message Catalog.

MessageBox can be used for simple informational display, where the user reads the message, then clicks an **OK** button to dismiss the message box. Use **MessageBox** as a way of branching based on user choice, in which case the message box contains two or more buttons (such as **OK** and **Cancel** or **Yes**, **No**, and **Cancel**). The value returned by the function tells you which button the user clicked, and your code can branch based on that value.

In the MessageBox dialogs, both the Text and the Explanation, that is, more detailed information stored in the Message Catalog, are included.

Note. In Windows Client, MessageBox dialogs include an **Explain** button to display more detailed information stored in the Message Catalog.

If **MessageBox** displays buttons other than **OK**, it causes processing to stop while it waits for user response. This makes it a "user think-time" function, restricting its use to certain PeopleCode events.

See Chapter 1, PeopleCode Built-in Functions, MsgGet , page 328; Chapter 1, PeopleCode Built-in Functions, MsgGetText , page 330 and Chapter 1, PeopleCode Built-in Functions, MsgGetExplainText , page 329.

Message Retrieval

MessageBox retrieves a message from the Message Catalog and substitutes the values of the parameters into the text message and explanation.

You can access and update the Message Catalog through PeopleTools Utilities, using the Message Catalog page located under the Use menu. You can enter message text in multiple languages. The *message_set* and *message_num* parameters specify the message to retrieve from the catalog. If the message is not found in the Message Catalog, the default message provided in *default_txt* is used. Message sets 1 through 19,999 are reserved for use by PeopleSoft applications. Message sets 20,000 through 32,767 can be used by PeopleSoft users.

The optional *paramlist* is a comma-separated list of parameters; the number of parameters in the list is arbitrary. The parameters are referenced in the message text using the % character followed by an integer corresponding to the position of the parameter in the *paramlist*. For example, if the first and second parameters in *paramlist* were &FIELDNAME and &USERNAME, they would be inserted into the message string as %1 and %2. To include a literal percent sign in the string, use %%; %\ is used to indicate an end-of-string and terminates the string at that point. This is generally used to specify fixed-length strings with trailing blanks.

Message Severity

MessageBox specifies processing for error handling functions based on the Message Severity of the message, which you can set in the Message Catalog. This enables you to change the severity of an error without changing the underlying PeopleCode, by setting the severity level for the message in the Message Catalog. The Message Severity settings and processing options are:

Severity	Processing
Message	The message is displayed and processing continues.
Warning	The message is displayed and treated as a warning.
Error	The message is displayed and treated as an error.
Cancel	The message is displayed and forces a Cancel.

In addition, in the PeopleSoft Internet Architecture the Message Severity dictates how the message displays:

- If the message has a severity of Warning, Error or Cancel, the message is displayed in a pop-up dialog box with a single **OK** button *regardless* of the value of the *style* parameter.
- If the message has a severity of Message and *style* is **%MsgStyle_OK** (0), the message displays in a pop-up dialog box with the single **OK** button.
- If the message has a severity of Message and *style* is not **%MsgStyle_OK** (0), the message displays in a separate window.

Restrictions on Use in PeopleCode Events

If **MessageBox** displays any buttons other than **OK**, it returns a value based on the end-user response and interrupts processing until the end-user has clicked one of the buttons. This makes it a "user think-time" function, subject to the same restrictions as other think-time functions which means that it cannot be used in any of the following PeopleCode events:

- SavePreChange.
- Workflow.
- RowSelect.
- SavePostChange.
- Any PeopleCode event that fires as a result of a ScrollSelect (or one of its relatives) function calls, or a Select (or one of its relatives) Rowset class method.

If the *style* parameter specifies a single button (that is, the **OK** button), the function can be called in any PeopleCode event.

See *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Think-Time Functions.

Restrictions on Use With PeopleSoft Internet Architecture

In the PeopleSoft Internet Architecture, you can't change the icon of a message box. You can change the number and type of buttons, and the default button, but the message always displays with the warning icon (a triangle with an exclamation mark in it.)

In addition, you can't change the message box title.

If the message has a severity of Warning and the MessageBox PeopleCode is in a SaveEdit event, the message is displayed in a new window with the OK and Cancel buttons.

Restrictions on Use With PeopleSoft Application Engine

If you call MessageBox from a PeopleCode action in an Application Engine program, the syntax is the same. However, all GUI-related parameters like style and title are ignored. You should use 0 and "".

Note. If you have an existing MessageBox in code called from a page, it should work as is.

The actual message data is routed to PS_MESSAGE_LOG at runtime, and you can view it from the Process Monitor by drilling down to the process details.

Parameters

Parameter	Description
Style	Either a numerical value or a constant specifying the contents and behavior of the dialog box. This parameter is calculated by cumulatively adding either a value or a constant from each of the following list of categories:

Note. In PeopleSoft Internet Architecture *style* is ignored if the message has any severity other than Message. If the message has a severity of Warning and the MessageBox PeopleCode is in a SaveEdit event, the message is displayed in a new window with the OK and Cancel buttons.

Category	Value	Constant	Meaning
Buttons	0	%MsgStyle_OK	The message box contains one pushbutton: OK.
	1	%MsgStyle_OKCancel	The message box contains two pushbuttons: OK and Cancel.
	2	%MsgStyle_ AbortRetryIgnore	The message box contains three pushbuttons: Abort, Retry, and Ignore.
	3	%MsgStyle_ YesNoCancel	The message box contains three pushbuttons: Yes, No, and Cancel.
	4	%MsgStyle_YesNo	The message box contains two push buttons: Yes and No.
	5	%MsgStyle_RetryCancel	The message box contains two push buttons: Retry and Cancel.

Note. The following values for *style* can only be used in Windows Client. They have no affect in PeopleSoft Internet Architecture.

Category	Value	Constant	Meaning
Default Button	0	%MsgDefault_First	The first button is the default.
	256	%MsgDefault_Second	The second button is the default.
	512	%MsgDefault_Third	The third button is the default.
Icon	0	%MsgIcon_None	None
	16	%MsgIcon_Error	A stop-sign icon appears in the message box.
	32	%MsgIcon_Query	A question-mark icon appears in the message box.
	48	%MsgIcon_Warning	An exclamation-point icon appears in the message box.
	64	%MsgIcon_Info	An icon consisting of a lowercase letter "I" in a circle appears in the message box.

Parameter	Description
title	Title of message box. If a null string is specified, then PeopleTools provides an appropriate value.

Note. *title* isn't available for messages displayed in the PeopleSoft Internet Architecture.

Parameter	Description
message_set	The message set number of the message to be displayed. When message set and number are provided, it overrides the specified text. A value less than one indicates that the message comes from the provided text and not the Message Catalog.
message_num	The message number of the message to be displayed.
default_txt	Default text to be displayed in the message box.
paramlist	A comma-separated list of parameters; the number of parameters in the list is arbitrary. The parameters are referenced in the message text using the % character followed by an integer corresponding to the position of the parameter in the *paramlist*.

Returns

Returns either a Number value or a constant. The return value is zero if there is not enough memory to create the message box. In other cases the following menu values are returned:

Value	Constant	Meaning
-1	%MsgResult_Warning	Warning was generated.
1	%MsgResult_OK	OK button was selected.
2	%MsgResult_Cancel	Cancel button was selected.
3	%MsgResult_Abort	Abort button was selected.
4	%MsgResult_Retry	Retry button was selected.
5	%MsgResult_Ignore	Ignore button was selected.
6	%MsgResult_Yes	Yes button was selected.
7	%MsgResult_No	No button was selected.

Note. In PeopleSoft Internet Architecture, pressing the ESC key has no effect. In Windows client, if a message box has a Cancel button, the same value is returned if either the ESC key is pressed or the Cancel button is selected. If the message box has no Cancel button, pressing ESC has no effect.

Example

Suppose the following string literal is stored in the Message Catalog as the message text:

```
Expenses of employee %1 during period beginning %2 exceed allowance.
```

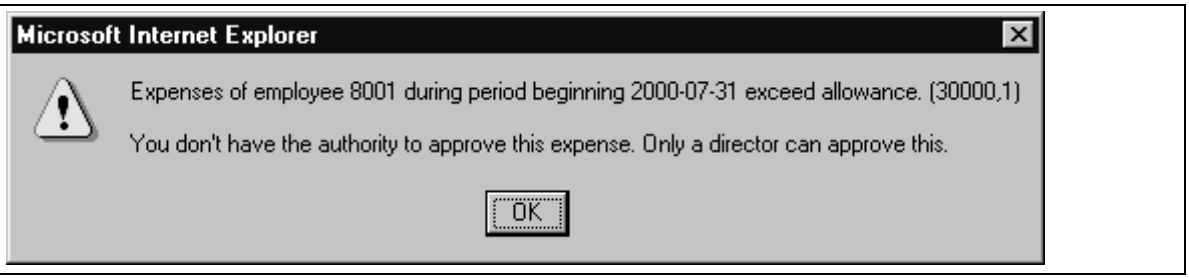
The following is stored in the Explanation:

```
You don’t have the authority to approve this expense. Only a director can approve⇒
 this.
```

Here %1 is a placeholder for the employee ID and %2 a placeholder for the expense period date. The following MessageBox call provides bind variables corresponding to these placeholders at the end of its parameter list:

```
MessageBox(0, "", 30000, 1, "Message not found.", BUS_EXPENSE_PER.EMPLID, ⇒
BUS_EXPENSE_PER.EXPENSE_PERIOD_DT);
```

The call would display a message box similar to this, if the message severity was Error or Warning.



Example of message box

Suppose the following is stored in the Message Catalog as the message text:

```
File not found.
```

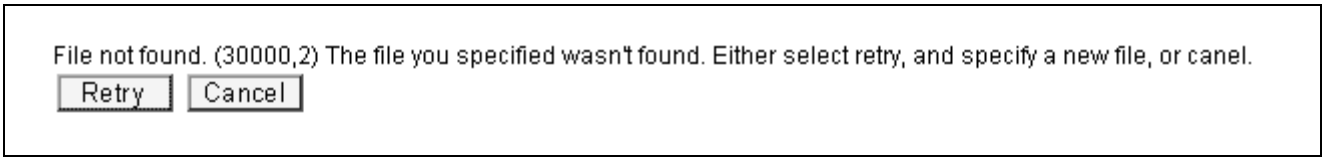
The following is stored in the Explanation:

```
The file you specified wasn’t found. Either select retry, and specify a new file,⇒
 or cancel.
```

Suppose this message had a Severity of message, and you used the %MsgStyle_ RetryCancel, in the following code:

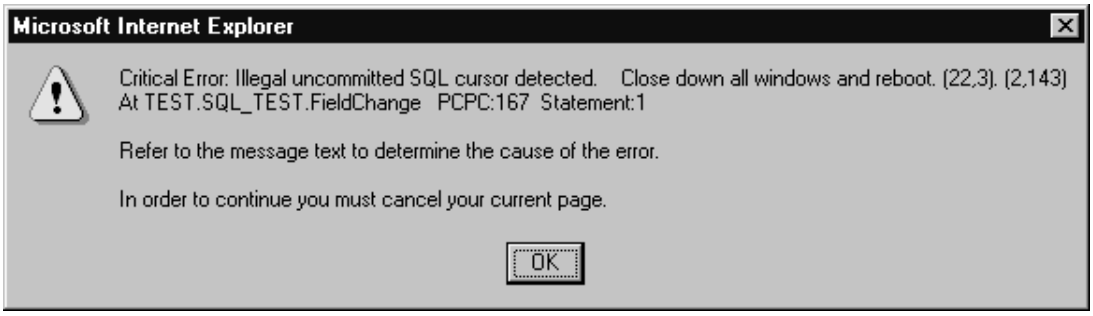
```
MessageBox(%MsgStyle_RetryCancel, "", 30000, 2, "Message not found.");
```

This is how the message displays:



Example message with Retry and Cancel buttons

If the message severity is of type Cancel, the message displayed looks like this:



Critical type error message

See Also

Chapter 1, PeopleCode Built-in Functions, MsgGet , page 328, Chapter 1, PeopleCode Built-in Functions, MsgGetText , page 330, Chapter 1, PeopleCode Built-in Functions, MsgGetExplainText , page 329.

Min

Syntax

Min(*param_list*)

Where *param_list* has the form

parameter1, *parameter2* [, *parameter3*, . . . *parameterN*]

Description

Use the **Min** function to determine the minimum value in the parameter list. The type of every item in the parameter list must be compatible with the first parameter in the list.

For example, if the first parameter is a string and the second parameter is a number with value 123.456, the second parameter is converted to the string "123.456" before the comparison is performed.

If all the values in the parameter list are alpha characters, "a" is less than "m", so Min("a", "m") returns "a".

Parameters

Parameter	Description
param_list	Specify a list of items to be compared. All items in the parameter list must be of the same type. If a value isn't defined, the system assumes it's of the same type as the first parameter.

Returns

The item in the list that has the minimum value.

Example

```
&RES = Min(&A, Max(&B, &C, &D), "-20");
```

See Also

Chapter 1, PeopleCode Built-in Functions, Max , page 319.

Minute

Syntax

Minute(*timevalue*)

Description

Use the **Minute** function to extract the minute component of a Time value.

Returns

Returns the minute part of *timevalue* as a Number data type.

Example

If &TIMEOUT contains "16:48:01" then the example sets &TIMEOUT_MINUTES to 48:

```
&TIMEOUT_MINUTES = Minute(&TIMEOUT);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Hour , page 291, Chapter 1, PeopleCode Built-in Functions, Second , page 393.

Mod

Syntax

Mod(*x*, *divisor*)

Description

Mod, the modulus function divides one number (*x*) by another (*divisor*) and returns the remainder.

Returns

Returns a Number equal to the remainder of the division of the number *x* by *divisor*.

Example

The example sets &NUM1 to one and &NUM2 to zero:

```
&NUM1 = Mod(10,3);
&NUM2 = Mod(10,2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Int , page 298, Chapter 1, PeopleCode Built-in Functions, Round , page 381, Chapter 1, PeopleCode Built-in Functions, Truncate , page 481.

Month

Syntax

Month(*datevalue*)

Description

Month returns the month of the year as an integer from 1 to 12 for the specified *datevalue*. The **Month** function accepts a Date or DateTime value as a parameter.

Parameters

Parameter	Description
datevalue	A Date or DateTime value on the basis of which to determine the month.

Returns

Returns a Number value from 1 to 12 specifying the month of the year.

Example

This example sets &HIRE_MONTH to 3:

```
&HIREDATE = DateTime6(1997, 3, 15, 10, 9, 20);
&HIRE_MONTH = Month(&HIRE_DATE);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date , page 117, Chapter 1, PeopleCode Built-in Functions, Date3 , page 118, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Day , page 127, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129, Chapter 1, PeopleCode Built-in Functions, Weekday , page 501, Chapter 1, PeopleCode Built-in Functions, Year , page 509.

MsgGet

Syntax

MsgGet(*message_set*, *message_num*, *default_msg_txt* [, *paramlist*])

where *paramlist* is an arbitrary-length list of parameters of undetermined (Any) data type to be substituted in the resulting text string, in the form:
param1 [, *param2*]. . .

Description

The **MsgGet** function retrieves a message from the PeopleCode Message Catalog and substitutes in the values of the parameters into the text message.

You can access and update the Message Catalog through the PeopleTools Utilities, using the Message Catalog page located under the Use menu. You can enter message text in multiple languages. The Message Catalog also enables you to enter more detailed "Explain" text about the message. The *message_set* and *message_num* parameters specify the message to retrieve from the catalog. If the message is not found in the Message Catalog, the default message provided in *default_msg_txt* is used. Message sets 1 through 19,999 are reserved for use by PeopleSoft applications. Message sets 20,000 through 32,767 can be used by PeopleSoft users.

The optional *paramlist* is a comma-separated list of parameters; the number of parameters in the list is arbitrary. The parameters are referenced in the message text using the % character followed by an integer corresponding to the position of the parameter in the *paramlist*. For example, if the first and second parameters in *paramlist* were &FIELDNAME and &USERNAME, they would be inserted into the message string as %1 and %2. To include a literal percent sign in the string, use %%; %\ is used to indicate an end-of-string and terminates the string at that point. This is generally used to specify fixed-length strings with trailing blanks.

MsgGet suffixes the message with "[Message Set# and Message Error#]", so it can be processed by a user not conversant in the translated language.

Example

```
&MsgText = MsgGet(30000, 2, "Message not found");
```

See Also

Chapter 1, PeopleCode Built-in Functions, MsgGetText , page 330, Chapter 1, PeopleCode Built-in Functions, MsgGetExplainText , page 329, Chapter 1, PeopleCode Built-in Functions, MessageBox , page 319.

MsgGetExplainText

Syntax

MsgGetExplainText(*message_set*, *message_num*, *default_msg_txt* [, *paramlist*])

where *paramlist* is an arbitrary-length list of parameters of undetermined (Any) data type to be substituted in the resulting text string, in the form:
param1 [, *param2*]. . .

Description

The **MsgGetExplainText** function retrieves the Explain text of a message from the PeopleCode Message Catalog and substitutes the values of the parameters in *paramlist* into the explain text. It returns the resulting message explain text as a String data type.

You can access and update the Message Catalog through the PeopleTools Utilities, using the Message Catalog. You can enter messages in multiple languages.

Message sets 1 through 19,999 are reserved for use by PeopleSoft applications. Message sets 20,000 through 32,767 can be used by PeopleSoft users.

Unlike **MsgGet**, **MsgGetExplainText** returns the message without a message set and message number appended to the message.

Parameters

Parameter	Description
message_set	Specify the message set to be retrieved from the catalog. This parameter takes a number value.
message_num	Specify the message number to be retrieved from the catalog. This parameter takes a number value.
default_msg_txt	Specify the text to be displayed if the message isn't found. This parameter takes a string value.
paramlist	Specify values to be substituted into the message explain text. The parameters listed in the optional *paramlist* are referenced in the message explain text using the % character followed by an integer referencing the position of the parameter in the function call. For example, if the first and second parameters in *paramlist* were &FIELDNAME and &USERNAME, they would be inserted into the message string as %1 and %2. **Note.** This substitution only takes place in message explain text when the **MsgGetExplainText** function is used. If you use a message box, the parameter substitution will *not* occur in the explain text. To include a literal percent sign in the string, use %%; %\ is used to indicate an end-of-string and terminates the string at that point. This is generally used to specify fixed-length strings with trailing blanks.

Example

Suppose the following explain text is stored in the Message Catalog:

```
A reference was made to a record.field (%1.%2) that is not defined within⇒
 Application Designer.  Check for typographical errors in the specification of the⇒
```

```
record.field or use Application Designer to add the new field or record.
```

Here %1 is a placeholder for the record name and %2 a placeholder for the field name. If the record.field in error was MyRecord.Field5, the above would resolve as follows:

```
A reference was made to a record.field (MyRecord.Field5) that is not defined⇒
 within Application Designer.  Check for typographical errors in the specification⇒
 of the record.field or use Application Designer to add the new field or record.
```

See Also

Chapter 1, PeopleCode Built-in Functions, MsgGetText , page 330, Chapter 1, PeopleCode Built-in Functions, MsgGet , page 328, Chapter 1, PeopleCode Built-in Functions, MessageBox , page 319.

MsgGetText

Syntax

MsgGetText(*message_set, message_num, default_msg_txt* [, *paramlist*])

where *paramlist* is an arbitrary-length list of parameters of undetermined (Any) data type to be substituted in the resulting text string, in the form:
param1 [, *param2*]. . .

Description

The **MsgGetText** function retrieves a message from the PeopleCode Message Catalog and substitutes the values of the parameters in *paramlist* into the text message. It returns the resulting message text as a String data type.

You can access and update the Message Catalog through the PeopleTools Utilities window, using the Message Catalog page located under the Use menu. You can enter message text in multiple languages. The *message_set* and *message_num* parameters specify the message to retrieve from the catalog. If the message is not found in the Message Catalog, the default message provided in *default_msg_txt* is used. Message sets 1 through 19,999 are reserved for use by PeopleSoft applications. Message sets 20,000 through 32,767 can be used by PeopleSoft users.

The parameters listed in the optional *paramlist* are referenced in the message text using the % character followed by an integer referencing the position of the parameter in the function call. For example, if the first and second parameters in *paramlist* were &FIELDNAME and &USERNAME, they would be inserted into the message string as %1 and %2. To include a literal percent sign in the string, use %%; %\ is used to indicate an end-of-string and terminates the string at that point. This is generally used to specify fixed-length strings with trailing blanks.

Unlike **MsgGet**, **MsgGetText** returns the message without a message set and message number appended to the message.

Example

```
&MsgText = MsgGetText(30000, 2, "Message not found");
```

See Also

Chapter 1, PeopleCode Built-in Functions, MsgGet , page 328, Chapter 1, PeopleCode Built-in Functions, MsgGetExplainText , page 329, Chapter 1, PeopleCode Built-in Functions, MessageBox , page 319.

NextEffDt

Syntax

NextEffDt(*field*)

Description

NextEffDt returns the value of the specified *field* from the record with the next effective date (and effective sequence number if specified). The return value is an Any data type. This function is valid only for effective-dated records.

If the next record doesn't exist, the statement is skipped. If the NextEffDt function isn't a top-level statement, that is, if it's contained within a compound statement or a loop, the statement is skipped and execution restarts with the *next* top-level statement.

In the following example, execution skips to the top If statement:

```
If ACTION <> "REH" Then  /* skip to here if NextEffDt fails to find next record */
   If    STD_HOURS <> NextEffDt(STD_HOURS) And
         Day(EFFDT) <> 1 Then
      Error MsgGet(30000, 8, "Meldung nicht vorhanden - WAZ bzw.⇒
 Beschäftigungsgradänderungen sind nur zum ersten eines Monats zulässig.")
   End-If;
End-If;
/* if NextEffDt fails, run to here directly */
```

See Also

Chapter 1, PeopleCode Built-in Functions, NextRelEffDt , page 331, Chapter 1, PeopleCode Built-in Functions, PriorRelEffDt , page 348, Chapter 1, PeopleCode Built-in Functions, PriorEffDt , page 347.

NextRelEffDt

Syntax

NextRelEffDt(*search_field*, *fetch_field*)

where *fieldlist* is an arbitrary-length list of fields in the form:
field1 [,*field2*]. . .

Description

NextRelEffDt locates the next occurrence of the *search_field* with the next effective date (and effective sequence number if the record contains an effective sequence number). It then returns the value of the specified *fetch_field* corresponding to the *search_field*. The return value is an Any data type. Typically, this function is used to retrieve values for related display fields.

This function is valid only for effective-dated records.

If a next record doesn't exist, the statement is skipped. If the NextRelEffDt function isn't a top-level statement, that is, if it's contained within a compound statement or a loop, the statement is skipped and execution restarts with the *next* top-level statement.

See Also

Chapter 1, PeopleCode Built-in Functions, GetRelField , page 266, Chapter 1, PeopleCode Built-in Functions, NextEffDt , page 331, Chapter 1, PeopleCode Built-in Functions, PriorRelEffDt , page 348, Chapter 1, PeopleCode Built-in Functions, PriorEffDt , page 347.

NodeDelete

Syntax

NodeDelete(*nodeName*)

Description

Use **NodeDelete** to delete the specified node and all subordinate objects (transactions, node properties, certificates, and so on.)

Warning! Once this function has completed, you cannot recover the node.

Event Considerations

PeopleSoft recommends only using this function in the SavePostChange event. In addition, you should put a warning in the SaveEdit event, so the user has a chance to change their mind about deleting the node.

If you use a pushbutton on a page to delete a node, PeopleSoft recommends the following code in the FieldChange event:

```
If %Page = Page.YourDeletePage Then
/* changes the record in the buffer so that the DoSaveNow fires */
   PSMSGNODEDEFN.DESCR = PSMSGNODEDEFN.DESCR | " ";
   DoSaveNow();
   ClearKeyList();
/* Transfer to another component or display information message; */
End-If;
```

Parameters

Parameter	Description
nodeName	Specify the name of the node you want to delete, as a string. All node names are uppercase.

Returns

A Boolean value: True, the function completed successfully deleted, False otherwise.

Example

```
&Rslt = NodeDelete("QEM_TEST_NODE");

If Not &Rslt Then

  /* Do error processing */

End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, NodeRename , page 333, Chapter 1, PeopleCode Built-in Functions, NodeSaveAs , page 333, *PeopleSoft Integration Broker*, Administering Basic Integrations.

NodeRename

Syntax

NodeRename(*oldNodeName, newNodeName*)

Description

Use **NodeRename** to rename a node. All node names are uppercase.

Event Considerations

PeopleSoft recommends using this function only in the SavePreChange event. This gives the user a chance to edit any other page fields before executing, which may be important because this function affects several tables.

Parameters

Parameter	Description
oldNodeName	Specify the name of the node that you want to change, as a string.
newNodeName	Specify the new name for the node, as a string.

Returns

A Boolean value: True, the function completed successfully deleted, False otherwise.

Example

```
&Rslt = NodeRename("QEM_TEST_NODE", "QE_TEST_NODE");

If Not &Rslt Then

  /* Do error processing */

End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, NodeDelete , page 332, Chapter 1, PeopleCode Built-in Functions, NodeSaveAs , page 333, *PeopleSoft Integration Broker*, Administering Basic Integrations.

NodeSaveAs

Syntax

NodeSaveAs(*oldNodeName, newNodeName*)

Description

Use **NodeSaveAs** to create a copy of the node specified by *oldNodeName*, and save it to the database as *newNodeName*. All node names are uppercase.

Event Considerations

PeopleSoft recommends using this function only in the SavePreChange event. This gives the user a chance to edit any other page fields before executing, which may be important because this function affects several tables.

Parameters

Parameter	Description
oldNodeName	Specify the name of the node that you want to copy, as a string.
newNodeName	Specify the name for the new node, as a string.

Returns

A Boolean value: True, the function completed successfully deleted, False otherwise.

Example

```
&Rslt = NodeSaveAs("PRODUCTION_NODE", "MY_TEST_NODE");

If Not &Rslt Then

  /* Do error processing */

End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, NodeRename , page 333, Chapter 1, PeopleCode Built-in Functions, NodeDelete , page 332, *PeopleSoft Integration Broker*, Administering Basic Integrations.

NodeTranDelete

Syntax

NodeTranDelete(*MsgNodeName, EffDt, TrxType, RqstMsgName, RqstMsgVer*);

Description

Use **NodeTranDelete** to delete a node transaction.

Warning! If you delete a node transaction, any transaction modifier using that transaction is also deleted.

Parameters

Parameter	Description
MsgNodeName	Specify the message node name as a string.
EffDt	Specify the effective date as a string.
TrxType	Specify the transaction type as a string.
RqstMsgName	Specify the request message name as a string.
RqstMsgVer	Specify the request message version as a string.

Returns

A Boolean value, True if the function completed successfully, False otherwise.

Example

```
&ret = NodeTranDelete("QE_LOCAL", "1900-01-01",  "IA",  "ROLESYNCH_MSG",  ⇒
"VERSION_1");
```

See Also

Chapter 1, PeopleCode Built-in Functions, RelNodeTranDelete , page 363.

None

Syntax

None(*fieldlist*)

where *fieldlist* is an arbitrary-length list of fields in the form:
[*recordname.*]*fieldname1* [, [*recordname.*]*fieldname2*] ...

Description

None takes an arbitrary number of field names as parameters and tests for values. **None** returns True if none of the specified fields contain a value. It returns False if any one of the fields contains a value.

A blank character field, or a zero (0) numeric value in a required numeric field is considered a null value.

Related Functions

All Checks to see if a field contains a value, or if all the fields in a list of fields contain values. If any of the fields is Null, then **All** returns False.

AllOrNone Checks if either all the field parameters have values, or none of them have values. Use this in cases where if an end-user fills in one field, she must all fill in the other related values.

OnlyOne Checks if exactly one field in the set has a value. Use this when the end-user must fill in only one of a set of mutually exclusive fields.

OnlyOneOrNone Checks if no more than one field in the set has a value. Use this in cases when a set of fields is both optional and mutually exclusive; that is, if the end-user puts can put a value into one field in a set of fields, or leave them all empty.

Example

The following example uses **None** to check whether REFERRAL_SOURCE has a value:

```
If None(REFERRAL_SOURCE) or
   REFERRAL_SOURCE = "EE" Then
   Gray(EMP_REFERRAL_ID);
End-if;
```

The following example uses **None** with a variable:

```
&ONETIME = FetchValue(POSN_INCUMB_WS.EMPLID, 1);
   If None(&ONETIME) Then
   /* do processing */
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, All , page 41, Chapter 1, PeopleCode Built-in Functions, AllOrNone , page 42, Chapter 1, PeopleCode Built-in Functions, OnlyOne , page 344, Chapter 1, PeopleCode Built-in Functions, OnlyOneOrNone , page 345.

NotifyQ

Syntax

NotifyQ(*logical queue ID*, *task type*)

Description

NotifyQ notifies the queue server of an incoming task. **NotifyQ** should *always* follow the use of **EnQueue**.**EnQueue** inserts the task into the PeopleSoft database, and NotifyQ notifies the queue server about a task's existence and location.

When you process a batch of tasks to be enqueued, PeopleSoft recommends calling **NotifyQ** just once (after the last task is processed). NotifyQ forces the queue server to reorder its internal lists and save its state to the database, and therefore affects performance.

NotifyQ is not required for chat or voice tasks, and should not be used for these tasks.

Note. If tasks of different types or tasks that are assigned to different logical queues are enqueued, at least one **NotifyQ** is required for each logical queue and for each task type. This ensures that the system is notified of tasks waiting to be assigned.

Parameters

Parameter	Description
logical queue ID	Specifies the logical queue in which the task should be queued. It is a string value. The logical queue ID is a case-sensitive value. The case used in the **NotifyQ** function must exactly match the case used when creating the logical queue ID on the Queues page.
task type	Specifies the type of task to be inserted. It is a string value. The valid values are: • email • generic

Returns

Returns 0 if the function was successful.

If unsuccessful, it returns a message number. The message set ID for MultiChannel Framework is 162.

For example, 1302 is returned when an invalid task type or no value is provided.

Example

```
&strtasknum = EnQueue(&queueID, "email", &MyCompURL, &langcode,
&subject, "QEDMO", 15, 60, &cost, &priority, &minskill);

&nret = NotifyQ(&queueID, "email");
   If &nret = 0 Then
         MessageBox(0, "", 162, 1547, "Queue Successfully notified.");

   End-If
```

NumberToString

Syntax

NumberToString(*Format*, *Number* [, *Width*] [, *Precision*])

Description

Use the **NumberToString** function to format *Number* according to the pattern specified in *Format*.

Specify the *Width* and *Precision* parameters when you want to dynamically specify the width or precision. Both width and precision can be set based on *Format*. For example, the following statically specifies the width to be 6, and the precision to be 2:

```
&MyValue = NumberToString("%6.2", &Num);
```

The following example show the width taken dynamically from the &Width variable:

```
&MyValue = NumberToString("%*.2", &Num, &Width);
```

The following example shows how both the width and the precision values are taken dynamically from the &Width and &Precision variables, respectively.

```
&MyValue = NumberToString("%*.*", &Num, &Width, &Precision);
```

Parameters

Parameter	Description
Format	Specify the pattern for of how *Number* is supposed to be formatted.
Number	Specify the *Number* to be formatted.
Width	Specify the width of the string to be formatted.
Precision	Specify the precision of the string to be formatted.

Using the Format Parameter

The *Format* parameter has the following format:

```
%[flags][width][.precision][R | T] [type]
```

- *Flags* have the following format:

Flag	Description
-	Left align the number.
$	Fill out field on left hand side with international currency symbol.
#	Force the number to have a decimal point.
blank	Pad left hand side with blanks only indicating a negative number with a ’-’ sign.
+	Prefix output with plus sign.
M	Append " (cr)" on right for negative numbers and " (dr)" for positive numbers.
m	Opposite of M: " (dr)" for negative and " (cr)" for positive.
A	Bracket negative numbers with "[" and "]".
a	Bracket negative numbers with "(" and ")".
q	Display zeros as blanks.

Flag	Description
Q	Display zeros as "None".
0	Pad left hand side with zeroes. This must be the last flag before any other pattern indicators.

- *Width* must be specified as a non-negative integer. Specifying an asterisks ("*") allows for dynamic field width specification. The maximum width is 50.
- *Precision* specifies how many digits follow the ".". It must be specified as a non-negative integer. Specifying an asterisks ("*") allows for a dynamic precision specification. The maximum precision is 50.
- *R* specifies rounding in conversion from the internal PeopleCode representation, that is, specifying 12.345 with precision of 2 (%n.2Rt) prints 12.35. In the absence of the R control rounding is the default.
- *T* specifies truncation in conversion from the internal PeopleCode representation, that is, specifying 2.345 with precision of 2 (%n.2Tt) prints 12.34.
- *Type* has the following format:

Type	Description
t	Type has format like printf %f. For example, the form dddd.dddd. This is the default value.
v	1000ths separator delimited output. For example, if the separator is a comma, the format is 1,000,000.02.
w	Scientific format like printf %e. For example, the form d.ddddeddd where "e" indicates exponent. d specifies 1 decimal digit and dddd specifies an arbitrary number.
W	Scientific format (like above, for "w") except "e" is "E".
z	Scientific Engineering format like printf %e where the exponent is always a multiple of 3 and the mantissa is between 1 and a 1000.
Z	Scientific Engineering format (like above, for "z") except "e" is "E".

Returns

A string value.

Example

In the following example, &Str1 would be "0001234.56".

```
&Num = 1234.56;

&Str1 = NumberToString("%#010.2t", &Num);
```

In the following example, &Str2 would be "$$$1234.56".

```
&Num = 1234.56;

&Str2 = NumberToString(("%$10.2", &Num);
```

In the following example, &Str3 would be " 1,234.56".

```
&Num = 1234.56;

&Str3 = NumberToString(("%10.2v", &Num);
```

In the following example, &Str4 would be "1.23456e+003".

```
&Num = 1234.56;

&Str4 = NumberToString(("%w", &Num);
```

See Also

Chapter 1, PeopleCode Built-in Functions, String , page 451, Chapter 1, PeopleCode Built-in Functions, Value , page 493.

ObjectDoMethod

Syntax

ObjectDoMethod(*obj_this*, *str_method_name* [, *paramlist*])

Where *paramlist* is a list of parameters of arbitrary length:
param1 [, *param2*]. . .

Description

Use the **ObjectDoMethod** function to invoke the method specified by *str_method_name* for the object *object_this,* passing in any required parameters using *paramlist*.

You can use ObjectDoMethod with Component Interfaces, Application Classes, OLE Automation objects, and so on.

This method can be useful if you know the number of parameters you need to pass for a method. If you don’t know how many parameters you may need to pass when you write your PeopleCode, use the **ObjectDoMethodArray** function.

Parameters

Parameter	Description
obj_this	Specify an already instantiated object. This variable must have been instantiated either with **CreateObject**, or another function or method that creates objects.
str_method_name	A string containing the name of an exposed method of *obj_this*.
paramlist	The parameter list to pass to the *str_method_name* method.

Returns

None.

Example

This simple example instantiates an Excel worksheet object, makes it visible, names it, saves it, and displays its name.

```
&WORKAPP = CreateObject("Excel.Application");
&WORKBOOKS = ObjectGetProperty(&WORKAPP, "Workbooks");
ObjectDoMethod(&WORKBOOKS, "Add", "C:\TEMP\INVOICE.XLT"); /* This associates the⇒
 INVOICE template w/the workbook */
ObjectDoMethod(&WORKAPP, "Save", "C:\TEMP\TEST1.XLS");
ObjectSetProperty(&WORKAPP, "Visible", True);
```

This simple example invokes a user-defined method associated with the current component interface object:

```
ObjectDoMethod(%CompIntfcName, &inMethodName);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateObject , page 101, Chapter 1, PeopleCode Built-in Functions, ObjectGetProperty , page 342, Chapter 1, PeopleCode Built-in Functions, ObjectSetProperty , page 343, Chapter 1, PeopleCode Built-in Functions, CreateObjectArray , page 103, Chapter 1, PeopleCode Built-in Functions, ObjectDoMethodArray , page 341, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using OLE Functions.

ObjectDoMethodArray

Syntax

ObjectDoMethodArray(*Object_Name, Method_Name, Array_of_Args*)

Description

Use the **ObjectDoMethodArray** function to invoke the method specified by *method_name* for the object *object_name*, passing in any required parameters using the array.

Use this function when you're not certain, at the time you're writing your PeopleCode program, how many parameters a method is going to require. If you know the number of parameters, use the **ObjectDoMethod** function instead.

The array of parameters is an array of Any. It can only be one-dimensional. You cannot pass in field references, that is, you can't pass in references of the form:

```
RECORD.FIELDNAME
```

If you don't want to supply any parameters, you can use an empty array, or a reference to a Null array.

Parameters

Parameter	Description
Object_Name	Specify an already instantiated object on which the method is to be evaluated.
Method_Name	Specify the name of an exposed method for the object.
Array_Of_Args	Specify an Array of Any containing the parameters for the method.

Returns

Depends on the specified object and method if a result is returned or not.

Example

```
&MyRslt = ObjectDoMethodArray(&MyObject, "My-Method", &MyArray);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateObject , page 101, Chapter 1, PeopleCode Built-in Functions, ObjectGetProperty , page 342, Chapter 1, PeopleCode Built-in Functions, ObjectSetProperty , page 343, Chapter 1, PeopleCode Built-in Functions, CreateObjectArray , page 103, Chapter 1, PeopleCode Built-in Functions, ObjectDoMethod , page 340, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using OLE Functions, Chapter 7, Array Class , page 633.

ObjectGetProperty

Syntax

ObjectGetProperty(*obj_this*, *str_property_name* [, *index_param_list*])

Description

ObjectGetProperty returns the value of a property *str_property_name* of the object *obj_this*.

Note. The object must have already been instantiated, either using **CreateObject** or another function or method that returns an object. Default" OLE Automation object properties are not supported. You must specify the object property that you want to retrieve explicitly.

Parameters

Parameter	Description
obj_this	Specify an already instantiated object. This variable must have been instantiated either with **CreateObject** or another function or method that creates objects.
str_property_name	A string containing the name of an exposed property of *obj_this*.
index_param_list	A comma-separated list for accessing an OLE automation object indexed property. (These parameters are only used with OLE/COM objects.)

Returns

Returns an Any value equal to the value of the *str_property_name* property of the *obj_this* object.

Example

This simple example instantiates an Excel worksheet object, makes it visible, names it, saves it, and displays its name.

```
&WORKAPP = CreateObject("Excel.Application");
&WORKBOOKS = ObjectGetProperty(&WORKAPP, "Workbooks");
ObjectDoMethod(&WORKBOOKS, "Add", "C:\TEMP\INVOICE.XLT"); /* This associates the⇒
 INVOICE template w/the workbook */
ObjectDoMethod(&WORKAPP, "Save", "C:\TEMP\TEST1.XLS");
ObjectSetProperty(&WORKAPP, "Visible", True);
```

Excel Worksheets had an index property called Range that has the following signature:

```
Property Range (Cell1 [, Cell2]) as Range
```

In the following example, the range is A1:

```
&CELL = ObjectGetProperty(&SHEET, "Range", "A1");
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateObject , page 101, Chapter 1, PeopleCode Built-in Functions, ObjectDoMethod , page 340, Chapter 1, PeopleCode Built-in Functions, ObjectSetProperty , page 343, Chapter 1, PeopleCode Built-in Functions, CreateObjectArray , page 103, Chapter 1, PeopleCode Built-in Functions, ObjectDoMethodArray , page 341, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using OLE Functions.

ObjectSetProperty

Syntax

ObjectSetProperty(*obj_this*, *str_property_name*, *val* [, *index_param_list*])

Description

ObjectSetProperty sets the value of a property *str_property_name* of the object *obj_this* to *val*.

The object must have already been instantiated, either using **CreateObject** or another function or method that returns an object.

Note. Default" OLE Automation object properties are not supported. You must specify the object property that you want to set explicitly.

Parameters

Parameter	Description
obj_this	Specify an already instantiated object. This variable must have been instantiated either with **CreateObject** or another function or method that creates objects.
str_property_name	A string containing the name of an exposed property of *obj_this*.
val	*str_property_name* is set to this value.
index_param_list	A comma-separated list of parameters for accessing an OLE automation object indexed property. (This is only used with COM/OLE objects.)

Returns

None.

Example

This simple example instantiates an Excel worksheet object, makes it visible, names it, saves it, and displays its name.

```
&WORKAPP = CreateObject("Excel.Application");
&WORKBOOKS = ObjectGetProperty(&WORKAPP, "Workbooks");
ObjectDoMethod(&WORKBOOKS, "Add", "C:\TEMP\INVOICE.XLT"); /* This associates the⇒
 INVOICE template w/the workbook */
ObjectDoMethod(&WORKAPP, "Save", "C:\TEMP\TEST1.XLS");
ObjectSetProperty(&WORKAPP, "Visible", True);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateObject , page 101, Chapter 1, PeopleCode Built-in Functions, ObjectDoMethod , page 340, Chapter 1, PeopleCode Built-in Functions, ObjectGetProperty , page 342, Chapter 1, PeopleCode Built-in Functions, CreateObjectArray , page 103, Chapter 1, PeopleCode Built-in Functions, ObjectDoMethodArray , page 341, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using OLE Functions.

OnlyOne

Syntax

OnlyOne(*fieldlist*)

where *fieldlist* is an arbitrary-length list of fields in the form:
[*recordname.*]*fieldname1* [, [*recordname.*]*fieldname2*] ...

Description

OnlyOne checks a list of fields and returns True if one and only one of the fields has a value. If all of the fields are empty, or if more than one of the fields has a value, OnlyOne returns False. This function is used to validate that only one of a set of mutually exclusive fields has been given a value.

A blank character field, or a zero numeric value in a required numeric field is considered a Null value.

Related Functions

All	Checks to see if a field contains a value, or if all the fields in a list of fields contain values. If any of the fields is Null, then **All** returns False.
None	Checks that a field or list of fields have no value. **None** is the opposite of **All**.
AllOrNone	Checks if either all the field parameters have values, or none of them have values. Use this in cases where if an end-user fills in one field, she must all fill in the other related values.
OnlyOneOrNone	Checks if no more than one field in the set has a value. Use this in cases when a set of fields is both optional and mutually exclusive; that is, if the end-user puts can put a value into one field in a set of fields, or leave them all empty.

Example

You typically use **OnlyOne** as follows:

```
If OnlyOne(param_one, param_two)
   Then value_a = "y";
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, All , page 41, Chapter 1, PeopleCode Built-in Functions, AllOrNone , page 42, Chapter 1, PeopleCode Built-in Functions, None , page 335, Chapter 1, PeopleCode Built-in Functions, OnlyOneOrNone , page 345.

OnlyOneOrNone

Syntax

OnlyOneOrNone(*fieldlist*)

where *fieldlist* is an arbitrary-length list of fields in the form:
[*recordname.*]*fieldname1* [, [*recordname.*]*fieldname2*] ...

Description

OnlyOneOrNone checks a list of fields and returns True if either of these conditions is true:

- Only one of the fields has a value.
- None of the fields has a value.

This function is useful when you have a set of mutually exclusive fields in a page and the entire set of fields is optional. The end-user can leave all the fields blank or enter a value in one of the fields only.

A blank character field, or a zero numeric value in a required numeric field is considered a null value.

Related Functions

All	Checks to see if a field contains a value, or if all the fields in a list of fields contain values. If any of the fields is Null, then **All** returns False.
None	Checks that a field or list of fields have no value. None is the opposite of All.
AllOrNone	Checks if either all the field parameters have values, or none of them have values. Use this in cases where if an end-user fills in one field, she must fill in all the other related values.
OnlyOne	Checks if exactly one field in the set has a value. Use this when the end-user must fill in only one of a set of mutually exclusive fields.

Example

You typically use **OnlyOneOrNone** as follows:

```
If OnlyOneOrNone(param_one, param_two)
   Then value_a = "y";
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, All , page 41, Chapter 1, PeopleCode Built-in Functions, AllOrNone , page 42, Chapter 1, PeopleCode Built-in Functions, None , page 335, Chapter 1, PeopleCode Built-in Functions, OnlyOne , page 344.

PanelGroupChanged

Syntax

PanelGroupChanged()

Description

Use **PanelGroupChanged** to determine whether a component has changed since the last save, whether by the user or by PeopleCode.

Note. The **PanelGroupChanged** function is supported for compatibility with previous releases of PeopleTools. New applications should use **ComponentChanged** instead.

See Also

Chapter 1, PeopleCode Built-in Functions, ComponentChanged , page 75.

PingNode

Syntax

PingNode(*MsgNodeName*)

Description

The **PingNode** function pings the specified node. It returns an array of numbers, containing the status codes for all locations. The status codes are ordered by SEQNO of the URL.

Parameters

Parameter	Description
MsgNodeName	Specify the message node you want to ping.

Returns

An array of data type number, containing the status codes for all locations. A status of 200 means the ping completed successfully. Any other status means ping did not complete successfully.

Example

```
Local array of number &StatusCodes;

&StatusCodes = PingNode(&MyMsgNode);
For &I = 1 to &StatusCodes.Len
   If &StatusCodes[&I] = 200 then;
      /* ping was successful – do some processing */
   Else
      /* do error processing */
   End-If;
      End-For;
```

See Also

Chapter 7, Array Class , page 633, Chapter 17, Message Class , page 973, *PeopleSoft Integration Broker*, Defining Message Channels and Messages.

PriorEffDt

Syntax

PriorEffDt(*field*)

Description

PriorEffDt returns the value of the specified field from the record with the prior effective date. This function is valid only for effective-dated records.

If the record contains an effective sequence number field, the value of this field is compared along with the effective-date field when the prior effective date/effective record sequence is determined. Therefore, if there is an effective-sequence number, it's possible that the effective-date field will be the same as the current record, but the sequence number would be earlier.

If a prior record does not exist, the statement is skipped. If the PriorEffDt function is not a top-level statement, that is, if it's contained with a compound statement or a loop, the statement is skipped and execution begins with the next top-level statement.

In the following example, execution skips to the top If statement:

```
If ACTION <> "REH" Then  /* skip to here if PriorEffdt fails to find prior record⇒
 */
   If    STD_HOURS <> PriorEffdt(STD_HOURS) And
         Day(EFFDT) <> 1 Then
      Error MsgGet(30000, 8, "Meldung nicht vorhanden - WAZ bzw.⇒
 Beschäftigungsgradänderungen sind nur zum ersten eines Monats zulässig.")
   End-If;
End-If;
/* if PriorEffdt fails, run to here directly */
```

Example

```
If CURRENCY_CD = PriorEffdt(CURRENCY_CD) Then
      Evaluate ACTION
      When = "PAY"
         If ANNUAL_RT = PriorEffdt(ANNUAL_RT) Then
            Warning MsgGet(1000, 27, "Pay Rate Change action is chosen and Pay⇒
 Rate has not been changed.");
         End-if;
         Break;
      When = "DEM"
         If ANNUAL_RT >= PriorEffdt(ANNUAL_RT) Then
            Warning MsgGet(1000, 29, "Demotion Action is chosen and Pay Rate has⇒
 not been decreased.");
         end-if;
      When-other
      End-evaluate;
      WinMessage("This message appears after executing either of the BREAK⇒
 statements or after all WHEN statements are false");
   End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, NextEffDt , page 331, Chapter 1, PeopleCode Built-in Functions, NextRelEffDt , page 331, Chapter 1, PeopleCode Built-in Functions, PriorRelEffDt , page 348.

PriorRelEffDt

Syntax

PriorRelEffDt(*search_field*, *fetch_field*)

Description

PriorRelEffDt locates the prior occurrence of the *search_field* with the prior effective date (and effective-sequence number if specified) and then returns the value of the specified *fetch_field* corresponding to the *search_field*. The return value is an Any data type. Typically, this function is used to retrieve values for related display fields.

This function is valid only for effective-dated records.

If a prior record does not exist, then the statement is skipped. If the PriorRelEffDt function isn't a top-level statement, that is, if it's contained within a compound statement or a loop, the statement is skipped and execution restarts with the *next* top-level statement.

See Also

Chapter 1, PeopleCode Built-in Functions, NextEffDt , page 331, Chapter 1, PeopleCode Built-in Functions, NextRelEffDt , page 331, Chapter 1, PeopleCode Built-in Functions, PriorEffDt , page 347.

PriorValue

Syntax

PriorValue(*fieldname*)

Description

PriorValue is used in FieldEdit and FieldChange PeopleCode to obtain the prior value of a buffer field that the user just changed. It returns the value that was in the buffer field before the user changed it, *not* the value of the field the last time the component was saved.

PriorValue gives correct results only in FieldEdit and FieldChange PeopleCode, and only for the buffer field where the function is executing. If you pass another field name to the function, it returns the current value of the buffer field, not the prior value.

Parameters

Parameter	Description
fieldname	The name of the record field. For correct results, this must be the name of the field where the call to PriorValue executes.

Returns

Returns an Any value equal to the value that was in the current buffer field immediately prior to the last edit.

Example

The following example from FieldChange PeopleCode gets the prior value of a field:

```
&PRIOR = PriorValue(QUANTITY);
DERIVED_TEST.TOTAL_AMT = (DERIVED_TEST.TOTAL_AMT  -  &PRIOR) + QUANTITY;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CurrentRowNumber , page 116.

Product

Syntax

Product(*numlist*)

where *numlist* is an arbitrary-length list of numbers in the form

n1 [, *n2*]. . .

Description

Product multiplies all the numbers in *numlist* and returns the product as a Number data type. The numbers in the list can be any number expressed as a number, variable, or expression.

Returns

Returns a Number value equal to the product of the numbers in *numlist*.

Example

The example sets &N2 to 96:

```
&N2 = Product(4,80,0.3);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Fact , page 179.

Prompt

Syntax

Prompt(*title*, *heading*, {*fieldlist* | *&Record*})

where *fieldlist* is an arbitrary-length list of fields in the form:
field1 [, *label1* [, *tempvar1*]] [, *field2* [, *label2* [, *tempvar2*]]]...

Note that the *label* parameter is required before the temporary variable.

Description

Prompt displays a page prompting the user to insert values into one or more text boxes. If the user cancels the page, any entered values are discarded and the function returns False. When the prompt page is displayed, the text boxes are initially filled with default values from the fields in *fieldlist*. The user can change the values in the text boxes, then if the user clicks OK, the values are placed either into the buffer for the appropriate field, or into a temporary variable, if a *tempvar* for that field is provided in the function call.

Prompt can also take a record object. This is primarily used with the Query classes, when running a query, to prompt the end-user for the prompt values of a query.

Prompt is a think-time function, and as such cannot be used during the PeopleCode attached to the following events:

- SavePreChange
- Workflow
- RowSelect
- SavePostChange
- Any PeopleCode event that fires as a result of a ScrollSelect (or one of its relatives) function calls, or a Select (or one of its relatives) Rowset class method.

Prompt should also not be called in any PeopleCode event that fires as a result of a **ScrollSelect** or its relatives, or a **Select** Rowset class method or its relatives.

See Also

PeopleCode Developer's Guide, Using Methods and Built-In Functions, Think-Time Functions.

Parameters

Parameter	Description
title	Used as the title for the page.
heading	Displayed in the page above the fields. If a zero-length string ("") is passed, the heading line is omitted in the page.
fieldlist | *&Record*	A list of one or more fields; each field in the list consists of a [*recordname*.]*fieldname* followed by an optional label and an optional temporary variable for storing the input value. The label parameter is required if you supply the temporary variable parameter. Instead of a list of fields, you can also specify an already instantiated and populated record object.
field	The name of the field being prompted for, the form [*recordname*.]*fieldname*.
label	Optional label for the prompted field. If this parameter is omitted, the field RFT Long value is used. This parameter is required before the *tempvar* parameter.
tempvar	Optional temporary variable to receive the user-entered value. If this parameter is omitted, the value is placed into the buffer for the field specified. Using a temp variable enables the PeopleCode program to inspect and process the entered value without affecting the buffer contents.

Returns

Optionally returns a Boolean value:

- False if the user clicks the Cancel button.
- True if the user clicks the OK button.

Example

The following example prompts for a single field, using calls to **MsgGetText** to retrieve values for the window title and prompt, and places the result into FISCAL_YEAR field:

```
Prompt(MsgGetText(5000, 182, " "), MsgGetText(5000, 184, " "), FISCAL_YEAR);
```

To following example places the results of the prompt into a temporary variable:

```
Prompt("Change Voucher", "", VOUCHER_ID, "Change Voucher ID to", &NEW_VOUCHER_ID);
```

The following code is in the USA push button FieldChange PeopleCode, and calls for the single field as shown in the page.

```
When = PAGE.PERSONAL_DATA1
        /* Administer Global Personnel - USA Flag Btn on PERSONAL_DATA1 Page */
         Prompt("US Social Security Number", "", PERSONAL_DATA.SSN);
         Break;
```

See Also

Chapter 24, Query Classes , page 1303.

Proper

Syntax

Proper(*string*)

Description

Proper capitalizes the first letter in a text *string* and any other letters in a text *string* that follow any character other than another letter. It also converts all other letters in a text *string* to lowercase.

Returns

Returns a String value with the first character of each word capitalized.

Example

The example sets the value of &BADKD to "K. D. Lang".

```
&BADKD = Proper("k.  d.  LANG")
```

See Also

Chapter 1, PeopleCode Built-in Functions, Lower , page 316, Chapter 1, PeopleCode Built-in Functions, Upper , page 492.

PublishXmlDoc

Syntax

PublishXmlDoc(*&XmlDoc*, **Message**.*MessageName* [, **Node**.*NodeName*])

Description

Use the **PublishXmlDoc** function to send an asynchronous message that is based on an XmlDoc object. The XmlDoc object must already be instantiated and populated. The message included in the function call should be an *unstructured* message, that is, one that isn't based on a hierarchical record structure.

If you want to retrieve an XmlDoc message that was sent asynchronously, use the GetMessageXmlDoc built-in function.

If you want to handle an XmlDoc as a Message object, you need to define a Message object with a hierarchical structure and migrate the data in the XmlDoc object into the Message object.

Parameters

Parameter	Description
&XmlDoc	Specify an already instantiated and populated XmlDoc object that you want to send as an asynchronous message.
MessageName	Specify an already existing unstructured message, prefaced with the reserved word **Message**.
NodeName	Specify a node. This is for Sender Specified Routing (SSR), prefixed with the reserved word **Node**. The node defines the target for the published message.

Returns

A Boolean value: True if the message was successfully published, False otherwise.

Example

```
Local XmlDoc &MyDoc;

. . .

PublishXmlDoc(&MyDoc, Message.MyXmlMessage, Node.MyNode);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetMessageXmlDoc , page 255, Chapter 1, PeopleCode Built-in Functions, SyncRequestXmlDoc , page 455.

PutAttachment

Syntax

PutAttachment(*URLDestination*, *SysFileName*, *LocalFile* [, *LocalDirEnvVar*])

where *URLDestination* can have *one* of the following forms:
URL.*URLname*

OR a string URL, such as
ftp://user:password@ftp.ps.com/

Description

The **PutAttachment** function transfers a file from the application sever to the file server.

Note. If the specified subdirectories don't exist this function tries to create them.

File Name Considerations

When the file is transferred, the following characters are replaced with an underscore:

- space
- semicolon
- plus sign
- percent sign
- ampersand
- apostrophe
- exclamation point
- pound sign
- dollar sign

Note. The @ sign is *not* changed to an underscore on files transferred with this function.

User Filenames for PutAttachment

If a user attaches a file through the browser dialog box generated by AddAttachment, the filename is converted *before* that filename is returned to the AddAttachment call for storage.

For example, "My Resume.doc" is returned through the AddAttachment parameter as "My_Resume.doc", with the space changed to an underscore.

No comparable mapping occurs for user filenames with PutAttachment, which requires the user filename as an input parameter, rather than an output parameter that is returned to the user.

Instead, before storing the ATTACHUSERFILE value, code similar to this should be invoked:

```
&ATTACHUSERFILE = Substitute(&ATTACHUSERFILE, " ", "_");
 &ATTACHUSERFILE = Substitute(&ATTACHUSERFILE, ";", "_");
 &ATTACHUSERFILE = Substitute(&ATTACHUSERFILE, "+", "_");
 &ATTACHUSERFILE = Substitute(&ATTACHUSERFILE, "%", "_");
 &ATTACHUSERFILE = Substitute(&ATTACHUSERFILE, "&", "_");
 &ATTACHUSERFILE = Substitute(&ATTACHUSERFILE, "'", "_");
 &ATTACHUSERFILE = Substitute(&ATTACHUSERFILE, "!", "_");
 &ATTACHUSERFILE = Substitute(&ATTACHUSERFILE, "@", "_");
 &ATTACHUSERFILE = Substitute(&ATTACHUSERFILE, "#", "_");
 &ATTACHUSERFILE = Substitute(&ATTACHUSERFILE, "$", "_");
```

File System Considerations

If you are uncertain which type of file system the file is going to transferred to, either a Unix or Windows system, you should use the last parameter (*LocalDirEnvVar*) and not specify a path to the file.

The following code example works for Windows systems, but *not* Unix systems:

```
PutAttachment(FTPINFO, TARGETFILENAME, "c:\temp\resume.doc");
```

The following code example works for Unix systems, but *not* Windows systems:

```
PutAttachment(FTPINFO, TARGETFILENAME, "/tmp/resume.doc");
```

The following two examples work for *both* Windows and Unix systems:

```
PutAttachment(FTPINFO, TARGETFILENAME, "resume.doc", "TMP");

PutAttachment(FTPINFO, TARGETFILENAME, "resume.doc", "PS_HOME");
```

Parameters

Parameter	Description
URLDestination	A reference to a URL. This can be either a URL name, in the form URL.*URLName*, or a string. This is where the file is transferred to.
SysFileName	The relative path and filename of the file on the file server. This is appended to *URLDestination* to make up the full URL where the file is transferred to. This parameter takes a string value.

Note. The *URLDestination* requires "/" slashes. Because *SysFileName* is appended to the URL, it also requires only "/" slashes. "\" are *not* supported for either the *URLDestination* or the *SysFileName* parameter.

Parameter	Description
LocalFile	The name, and possible full path, to the destination file on the application server. This is where the file is transferred from. This parameter takes a string value. If you only specify a file name, the file is assumed to be in the directory specified by the **TMP** environment variable or the directory specified by *LocalDirEnvVar*. If the file is *not* in the **TMP** directory and you don't specify *LocalDirEnvVar* you must use the full path name as appropriate to the operating system (that is, on Unix use "\").
LocalDirEnvVar	The name of an environment variable containing path information. If specified, the path value in the environment variable is prepended to *LocalFile*. Use this if you don't want to hardcode path names in the *LocalFile* parameter. This parameter takes a string value. **Note.** Note. Only operating system environment variables are recognized with this parameter, rather than variables defined in configuration files.

Returns

An integer value or a constant:

Numeric Value	Constant Value	Description
0	%Attachment_Success	File was transferred successfully.
1	%Attachment_Failed	File was not successfully transferred.
3	%Attachment_FileTransferFailed	File was not successfully transferred.
4	%Attachment_NoDiskSpaceAppServ	Transfer failed because no space was available on the application server.
5	%Attachment_NoDiskSpaceWebServ	Transfer failed because no space was available on the web server.
6	%Attachment_FileExceedsMaxSize	Transfer failed because the file was too large for the database platform. Check your database platform specifications.
7	%Attachment_DestSystNotFound	Transfer failed because the destination system wasn't found. Check the *URLDestination* parameter.
8	%Attachment_DestSysFailedLogin	Transfer failed because the system couldn't connect to the destination system specified with *URLDestination*.
9	%Attachment_FileNotFound	Transfer failed because the *LocalFile* couldn't be found.
10	%Attachment_DeleteFailed	Transfer failed because the file couldn't be deleted from application server.

Example

The following transfers the file "resume.txt" from the application server to the file server.

```
PutAttachment("ftp://anonymous:hobbit1@ftp.ps.com/HRarchive/", "NewHire⇒
/11042000resume.txt", " "C:\NewHires\resume.txt".");
```

See Also

Chapter 1, PeopleCode Built-in Functions, AddAttachment , page 31, Chapter 1, PeopleCode Built-in Functions, DeleteAttachment , page 135, Chapter 1, PeopleCode Built-in Functions, GetAttachment , page 234, Chapter 1, PeopleCode Built-in Functions, ViewAttachment , page 494, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Understanding File Attachment Architecture.

Quote

Syntax

Quote(*String*)

Description

Use the **Quote** function to enclose a string in single quotation marks. This function also can be used to put quotation marks around SQL statements.

Parameters

Parameter	Description
String	Specify the string you want to enclose in single quotation marks.

Returns

The string specified by *String* enclosed in single quotation marks.

Example

The following code returns 'Bob':

```
&QuotedString = Quote("Bob");
```

The following code returns 'Bob''s' (two single quotes to represent the apostrophe)

```
&QuotedString = Quote("Bob's");
```

The following code returns '%'' OR USER ID LIKE ''%':

```
&QuotedString = Quote("%' OR USERID LIKE '%");
```

Radians

Syntax

Radians(*angle*)

Description

Radians converts the given angle from degree measurement to radian measurement.

Parameters

Parameter	Description
angle	The size of an angle in degrees.

Returns

The size of the given angle in radians.

Example

The following example returns the equivalent size in radians of an angle measuring **65.5** degrees:

```
&RADIAN_SIZE = Radians(65.5);
```

The following example returns the value of **pi**, that is, **180** degrees expressed as radians:

```
&PI = Radians(180);
```

Note. This example represents **pi** with a high degree of accuracy, but no computer system can represent irrational numbers exactly. Thus, the results of extended calculations based on **pi** have the potential for a cumulative reduction in precision.

See Also

Chapter 1, PeopleCode Built-in Functions, Acos , page 29, Chapter 1, PeopleCode Built-in Functions, Asin , page 45, Chapter 1, PeopleCode Built-in Functions, Atan , page 46, Chapter 1, PeopleCode Built-in Functions, Cos , page 91, Chapter 1, PeopleCode Built-in Functions, Cot , page 92, Chapter 1, PeopleCode Built-in Functions, Degrees , page 134, Chapter 1, PeopleCode Built-in Functions, Sin , page 439, Chapter 1, PeopleCode Built-in Functions, Tan , page 456.

Rand

Syntax

Rand( )

Description

Rand generates a random number greater than or equal to 0 and less than 1. To generate a random integer that is greater than or equal to 0 but less than x, use **Int**(**Rand()***x*).

Returns

Returns a random Number value greater than or equal to 0 and less than 1.

Example

The example sets &RANDOM_NUM to a random value less than 100.

```
&RANDOM_NUM = Int(Rand( )*100)
```

See Also

Chapter 1, PeopleCode Built-in Functions, Int , page 298.

RecordChanged

Syntax

The syntax of **RecordChanged** varies, depending on whether you use a scroll path reference or a contextual reference to designate the row being tested.

Using a scroll path reference, the syntax is:
RecordChanged(*scrollpath, target_row*)

where *scrollpath* is:
[**RECORD.**level1_recname, level1_row, [**RECORD.**level2_recname, level2_row,]]
RECORD.target_recname

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Using a contextual reference the syntax is:
RecordChanged(**RECORD.***target_recname*)

A contextual reference specifies the current row on the scroll level designated by **RECORD.** *target_recname*.

An older construction, in which a record field expression is passed, is also supported. The record field is any field in the row where the PeopleCode program is executing (typically the one on which the program is executing).
RecordChanged(*recordname.fieldname*)

Description

RecordChanged determines whether the data in a specific row has been modified since it was retrieved from the database either by the user or by a PeopleCode program.

Note. This function remains for backward compatibility only. Use the **IsChanged** Record class property instead.

This is useful during save processing for making updates conditional on whether rows have changed.

Note. The word "record" is used in this function name in a misleading way. Remember that this function (like the related functions **RecordDeleted** and **RecordNew**) checks the state of a *row*, not a *record*.

See Also

Chapter 25, Record Class , IsChanged, page 1461, Chapter 26, Row Class , DeleteEnabled, page 1472,
PeopleCode Developer's Guide, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Understanding Current Context.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
RECORD .*target_recname*	The primary scroll record of the scroll level where the row being referenced is located. As an alternative, you can use **SCROLL.** *scrollname.*

Returns

Returns a Boolean value:

- True if any data in the target row has been changed.
- False if no data in the target row has been changed.

Example

This example shows a **RecordChanged** call using a contextual reference:

```
If RecordChanged(RECORD.BUS_EXPENSE_DTL) Then
   WinMessage("Changed row msg from current row.", 64);
End-If;
```

The following example, which would execute on level one, checks rows on level two to determine which have been changed:

```
For &I = 1 To ActiveRowCount(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1),⇒
 RECORD.BUS_EXPENSE_DTL);
   If RecordChanged(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1), ⇒
RECORD.BUS_EXPENSE_DTL, &I) Then
      WinMessage("Changed row message from level one.", 64);
   End-If;
End-For;
```

See Also

Chapter 1, PeopleCode Built-in Functions, FieldChanged , page 182, Chapter 1, PeopleCode Built-in Functions, RecordDeleted , page 359, Chapter 1, PeopleCode Built-in Functions, RecordNew , page 361.

RecordDeleted

Syntax

The syntax of **RecordDeleted** varies, depending on whether you use a scroll path reference or a contextual reference to designate the row being tested.

Using a scroll path reference, the syntax is:
RecordDeleted(*scrollpath, target_row*)

where *scrollpath* is:
[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]]
RECORD.*target_recname*

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Using a contextual reference the syntax is:
RecordDeleted(**RECORD.***target_recname*)

A contextual reference specifies the current row on the scroll level designated by **RECORD.** *target_recname.*

An older construction, in which a record field expression is passed, is also supported. The record field is any field in the row where the PeopleCode program is executing (typically the one on which the program is executing).

RecordDeleted(*recordname.fieldname*)

Description

RecordDeleted checks to see if a row of data has been marked as deleted, either by an end-user row delete (F8) or by a call to **DeleteRow**.

Note. This function remains for backward compatibility only. Use the **IsDeleted** Record class property instead.

RecordDeleted is useful during save processing to make processes conditional on whether a row has been deleted.

Deleted rows are not actually removed from the buffer until after the component has been successfully saved, so you can check for deleted rows all the way through SavePostChange PeopleCode.

RecordDeleted is not typically used in a loop, because it is easier to put it on the same scroll level as the rows being checked in SavePreChange or SavePostChange PeopleCode: these events execute PeopleCode on every row in the scroll, so no looping is necessary.

Note. To avoid confusion, note that this function (like the related functions **RecordChanged** and **RecordNew**) checks the state of a *row*, not a *record.*

See Also

Chapter 25, Record Class , IsDeleted, page 1462, Chapter 26, Row Class , IsDeleted, page 1473, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Understanding Current Context.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
RECORD. *target_recname*	The primary scroll record of the scroll level where the row being referenced is located. As an alternative, you can use **SCROLL.** *scrollname.*

Returns

Returns a Boolean value:

- True if the target row has been deleted.
- False if the target row has not been deleted.

Example

This example shows a **RecordDeleted** call using a contextual reference

```
If RecordDeleted(RECORD.BUS_EXPENSE_DTL) Then
   WinMessage("Deleted row msg from current row.", 64);
End-If;
```

The following example, which would execute on level zero, checks rows on level one to determine which have been deleted:

```
For &I = 1 To TotalRowCount(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1),⇒
 RECORD.BUS_EXPENSE_DTL);
   If RecordDeleted(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1), ⇒
RECORD.BUS_EXPENSE_DTL, &I) Then
      WinMessage("Deleted row message from level one.", 64);
   End-If;
End-For;
```

Note that the loop is delimited by **TotalRowCount**. **For** loops delimited by **ActiveRowCount** don't process deleted rows.

See Also

Chapter 1, PeopleCode Built-in Functions, FieldChanged , page 182, Chapter 1, PeopleCode Built-in Functions, RecordChanged , page 358, Chapter 1, PeopleCode Built-in Functions, RecordNew , page 361, Chapter 1, PeopleCode Built-in Functions, TotalRowCount , page 462, Chapter 1, PeopleCode Built-in Functions, ActiveRowCount , page 29.

RecordNew

Syntax

The syntax of **RecordNew** varies, depending on whether you use a scroll path reference or a contextual reference to designate the row being tested.

Using a scroll path reference, the syntax is:

RecordNew(*scrollpath, target_row*)

where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]]
RECORD.*target_recname*

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Using a contextual reference the syntax is:

RecordNew(**RECORD.***target_recname*)

A contextual reference specifies the current row on the scroll level designated by **RECORD.** *target_recname.*

An older construction, in which a record field expression is passed, is also supported. The record field is any field in the row where the PeopleCode program is executing (typically the one on which the program is executing).

RecordNew(*recordname.fieldname*)

Description

RecordNew checks a specific row to determine whether it was added to the component buffer since the component was last saved.

Note. This function remains for backward compatibility only. Use the **IsNew** Row class property instead.

This function is useful during save processing to make processes conditional on whether or not a row is new.

Note. To avoid confusion, remember that this function (like the related functions **RecordChanged** and **RecordDeleted**) checks the state of a *row*, not a *record.* In normal PeopleSoft usage, the word "record" denotes a table-level object (such as a table, view, or Derived/Work record).

See Also

Chapter 26, Row Class , IsNew, page 1474, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Understanding Current Context.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
RECORD. *target_recname*	The primary scroll record of the scroll level where the row being referenced is located. As an alternative, you can use **SCROLL.** *scrollname.*

Returns

Returns a Boolean value:

- True if the target row is new.
- False if the target row is not new.

Example

This example shows a **RecordNew** call using a contextual reference:

```
If RecordNew(RECORD.BUS_EXPENSE_DTL) Then
   WinMessage("New row msg from current row.", 64);
End-If;
```

The following example, which would execute on level one, checks rows on level two to determine which have been added:

```
For &I = 1 To ActiveRowCount(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1),⇒
 RECORD.BUS_EXPENSE_DTL);
   If RecordNew(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1), ⇒
RECORD.BUS_EXPENSE_DTL, &I) Then
      WinMessage("New row message from level one.", 64);
   End-If;
End-For;
```

See Also

Chapter 1, PeopleCode Built-in Functions, FieldChanged , page 182, Chapter 1, PeopleCode Built-in Functions, RecordChanged , page 358, Chapter 1, PeopleCode Built-in Functions, RecordDeleted , page 359.

RefreshTree

Syntax

RefreshTree(**Record**.*bound_recname*)

Description

RefreshTree updates a dynamic tree.

Note. Dynamic tree controls have been deprecated. Use **GenerateTree** or Tree Viewer.

See Also

Chapter 1, PeopleCode Built-in Functions, GenerateTree , page 230, Appendix B, Appendix B: Viewing Trees From Application Pages , page 1939.

RelNodeTranDelete

Syntax

RelNodeTranDelete(*RelationshipId , SrcTrxType, SrcNode, SrcRqstMsgName, SrcRqstMsgVer, TgtNode, TgtRqstMsgName, TgtRqstMsgName, TgtRqstMsgVer*);

Description

Use **RelNodeTranDelete** to delete a transaction modifier.

Parameters

Parameter	Description
RelationshipId	Specify the relationship ID as a string.
ScrTrxType	Specify the source transaction type as a string.
SrcNode	Specify the source node as a string.
ScrRqstMsgName	Specify the source request message name as a string.
ScrRqstMsgVer	Specify the source request message version as a string.
TgtNode	Specify the target node as a string.
TgtRqstMsgName	Specify the target request message name as a string.
TgtRqstMsgName	Specify the target message name as a string.
TgtRqstMsgVer	Specify the target request message version as a string.

Returns

A Boolean value, True if the function completed successfully, False otherwise.

Example

```
&ret = RelNodeTranDelete("QE_TEST", "CMS_TEST", "CMS_TEST_LOCAL", "OA", ⇒
 "ROLESYNCH_MSG",  "VERSION_1", "CMS_TEST_LOCAL2",   "ROLESYNCH_MSG2", ⇒
```

```
"VERSION_1",);
```

See Also

Chapter 1, PeopleCode Built-in Functions, NodeTranDelete , page 334.
PeopleTools PeopleBooks: Integration Broker, "Administering Basic Integrations," Configuring Transactions

RemoteCall

Syntax

RemoteCall(*dispatcher_name* [, *service_paramlist*] [, *user_paramlist*])

where *service_paramlist* and *user_paramlist* are arbitrary-length lists of parameters in the form:
var1, *val1* [, *var2*, *val2*]. . .

Description

RemoteCall is a PeopleCode API function that provides a means of calling a Tuxedo service from a PeopleSoft application. A typical use of Remote Call is to run data-intensive, performance-sensitive programs near or on the database server.

Note. For PeopleTools 8 you can no longer use RemoteCall to start an Application Engine program. You must use **CallAppEngine** instead.

Because complex PeopleCode processes can now be run on the application server in three-tier mode, the **RemoteCall** PeopleCode function has more limited utility. However, RemoteCall can still be very useful, because it provides a way to take advantage of existing COBOL processes.

- In three-tier mode, **RemoteCall** always runs on the application server.
- In two-tier mode, **RemoteCall** always runs on the client.

This means that it is no longer necessary to set a location for the remote call in PeopleSoft Configuration Manager.

Each RemoteCall service can have zero or more standard parameters and any number of user parameters. The standard parameters are determined by the RemoteCall dispatcher, the user parameters by the COBOL program being run.

There is only one RemoteCall dispatcher delivered with PeopleTools 7, PSRCCBL, which executes a COBOL program using the connect information of the current end-user.

RemoteCall can be used from any type of PeopleCode *except* SavePostChange, SavePreChange, Workflow, and RowSelect. However, remote programs that change data should not be run as part of the SaveEdit process, because the remote program may complete successfully even though an error occurs in a later part of the save process. For remote programs that change data, the normal place for them would be in the FieldChange PeopleCode behind a command push button, or in a pop-up menu item.

After you use **RemoteCall**, you may want to refresh your page. The **Refresh** method, on a rowset object, reloads the rowset (scroll) using the current page keys. This causes the page to be redrawn. The following code refreshes the entire page:

```
GetLevel0().Refresh()
```

If you only want a particular scroll to be redrawn, you can refresh just that part.

See Also

Chapter 1, PeopleCode Built-in Functions, CallAppEngine , page 58, Chapter 27, Rowset Class , Refresh, page 1495.

Parameters

The parameters passed to RemoteCall can be broken into three parts: the RemoteCall *Dispatcher Name*, the standard *Parameter Lists* for the service, and the *User Parameter Lists* for the program being called on the service.

Dispatcher Name

The *dispatcher_name* parameter is a string value that specifies the type of RemoteCall performed. For PeopleTools 7 there is only one RemoteCall dispatcher delivered, PSRCCBL, which executes a COBOL program using the connect information of the current end-user, so the value you pass to this parameter should always be "PSRCCBL". Future versions of PeopleTools may provide support for Red Pepper, SQR, or customer supplied remote calls.

Parameter Lists

Both the standard parameter list and user parameter list have the same form. Think of the parameters passed to the service as being passed as pairs of variable names and values of input and output parameters:

```
variable_name, value
```

Where:

- *variable_name* is a string literal or string variable that contains the name of the input or output variable as referenced in the remote program. For example, if the remote program expects a variable named "USERNAME", then the PeopleCode should use "USERNAME" or &VARIABLE (which had been assigned the value "USERNAME").
- For input variables, *value* is the value to be passed to the remote program with the variable name. It can be either a variable or literal with a data type that corresponds to the *variable_name* variable. For output variables, *value* is the value returned to the PeopleCode program from the remote program. It must be a variable in this case, representing the buffer into which the value is returned.

An arbitrary number of parameters can be passed to the service. There is, however, a limitation on the number of variables that can be passed in PeopleCode, which is limited by the size of the PeopleCode parameter stack, currently 128.

In the case of the PSRCCBL dispatcher, there are three standard parameters, listed in the following table:

Dispatcher	Parameter	Required	Description
PSRCCBL	PSCOBOLPROG	Y	Name of the COBOL program to run.

Dispatcher	Parameter	Required	Description
PSRCCBL	PSRUNCTL	N	Run-control parameter to pass to the COBOL program.
PSRCCBL	INSTANCE	N	Process instance parameter to pass to the COBOL program.

User Parameter List

For PSRCCBL, the remote COBOL program must match the user parameters to the usage of its application. The names of the parameters are sent to the server and can be used by the COBOL program. The COBOL program returns any modified (output) parameters by name. Parameters which are not returned are not modified, and any extra returned parameters (that is, parameters beyond the number passed or of different names) are discarded with no effect.

Returns

None.

Example

You could use the following PeopleCode to execute the program "CBLPROG1":

```
Rem Set the return code so we are sure it is sent back.
&Returncode = -1;
Rem Set the parameters that will be sent across.
&param1 = "John";
&param2 = "Smith";
Rem Set the standard parameters that indicate program name and run-control.
&RemoteCobolPgm = "CBLPROG1";
/* call the remote function */
RemoteCall ("PSRCCBL",
"PSCOBOLPROG", &RemoteCobolPgm,
"PSRUNCTL", workrec.runctl,
"FirstName", &param1,
"LastName", &param2,
"Returncode", &Returncode,
"MessageSet", &msgset,
"MessageID", &msgid,
"MessageText1", &msgtext1,
"MessageText2", &msgtext2);
if &Returncode <> 0
   WinMessage(MsgGet(&msgset, &msgid, "default message", &msgtext1, &msgtext2));
end-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Exec , page 171, Chapter 1, PeopleCode Built-in Functions, WinExec , page 503, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using RemoteCall.

RemoveDirectory

Syntax

```
RemoveDirectory(path [, remove_parameters])
```

where *remove_parameters* can be in the form:

```
path_type [+ directory_type]
```

Description

Use the **RemoveDirectory** function to remove the directory specified by *path*. You can also specify whether to remove just the directory, or to delete the directory and all subdirectories, including any files, that is, to remove the entire directory tree.

Working With Relative Paths

If you specify a relative path, that path is appended to the path specified by an environmental variable, if the variable exists, in a particular order. This means if the first environmental variable in order is found, that is the one that's used. If it isn't found, the next one is searched for, and if found, is used, and so on.

The environmental variables that are used are different, depending on whether the machine where the code is running is an application server. If the code is running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]

If the code is *not* running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]
3. TEMP[\relative_path]

The PS_FILEDIR environmental variable is *not* initialized and set to a path automatically. If you want to use this environmental variable, you must set it yourself.

Note. Your system security should verify if a user has the correct permissions before allowing access to a drive (for example, if a user changes their **TEMP** environmental variable to a network drive they don't normally have access to, your system security should detect this.)

Parameters

Parameter	Description
path	Specify the directory to be removed.
remove_parameters	Specify additional considerations about the directory to be removed. Specify whether the path is an absolute or relative path. Values are: • %FilePath_Relative (default) • %FilePath_Absolute If you don't specify *pathtype* the default is %FilePath_Relative. If the path is an absolute path, whatever path you specify is used verbatim. You must specify a drive letter as well as the complete path. You can't use any wildcards when specifying a path. The Component Processor automatically converts platform-specific separator characters to the appropriate form for where your PeopleCode program is executing. On a WIN32 system, UNIX "/" separators are converted to "\", and on a UNIX system, WIN32 "\" separators are converted to "/". Specify whether to remove only the specified directory or to remove the directory and all its subdirectories. The default is to just remove the specified directory. The valid values are: • %Remove_Subtree • %Remove_Directory **(default)**

Returns

None.

Example

The following example is for an NT operating system:

```
RemoveDirectory("C:\temp\mydir\temp", %filepath_absolute + %remove_subtree);
```

The following example is for a UNIX operating system:

```
RemoveDirectory("/temp/mydir/temp", %filepath_absolute + %remove_subtree);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateDirectory , page 96, Chapter 1, PeopleCode Built-in Functions, DeleteAttachment , page 135, Chapter 1, PeopleCode Built-in Functions, FileExists , page 184, Chapter 1, PeopleCode Built-in Functions, FindFiles , page 189, Chapter 13, File Class , page 873.

RenameDBField

Syntax

RenameDBField(**Field**.*NewFieldName*, **Field**.*OldFieldName*)

Description

Use the **RenameDBField** function to modify a field definition to have a new name. This function also cleans up most references, such as in PeopleCode programs and on records so they now use the new name.

Note. Because using this function changes records that are used to build application tables, you must rebuild (alter) the specified project before these changes can be used.

Considerations Using this Function

In SQL associated with records of type view, the field name is *not* changed. You must fix those by hand.

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

This operation is time consuming.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
NewFieldName	Specify the new field name to be used. This name must be prefixed by the reserved word **Field**.
OldFieldName	Specify the name of the field to be changed. This name must be prefixed by the reserved word **Field**.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Bulk operation completed successfully.
%MDA_Failure	Bulk operation did not complete successfully.
%MDA_FieldNotFound	The field specified by *OldFieldName* wasn't found in the specified project or page list.
%MDA_Duplicate	The field specified by *NewFieldName* already exists.

Example

```
&ret = RenameDBField(Field.OrgId, Field.DeptId);
If (&ret = %MDA_Success) Then
    MessageBox(0, "Metadata Fn Status", 0, 0, "RenameDBField succeeded");
Else
    MessageBox(0, "Metadata Fn Status", 0, 0, "RenameDBField failed");
End-If;
```

The following example de-references the field name for the function.

```
&oldcf = "CF1";
&newcf = "XYZ_STORE_ID";
```

```
&new = "FIELD." | &newcf;
&old = "FIELD." | &oldcf;
&ret = RenameDBField(@(&new), @(&old));
If (&ret = 0) Then
   MessageBox(0, "RenameDBField", 0, 0, "Succeeded");
Else
   MessageBox(0, "RenameDBField", 0, 0, "Failed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, RenamePage , page 370, Chapter 1, PeopleCode Built-in Functions, RenameRecord , page 371.

RenamePage

Syntax

RenamePage(**Page.***NewPageName*, **Page.***OldPageName*)

Description

Use the **RenamePage** function to modify a page definition to have a new name. This function also cleans up most references so they now use the new name.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

This operation is time consuming

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
NewPageName	Specify the new page name to be used. This name must be prefixed by the reserved word **Page**.
OldPageName	Specify the name of the page to be changed. This name must be prefixed by the reserved word **Page**.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Bulk operation completed successfully.

Value	Description
%MDA_Failure	Bulk operation did not complete successfully.
%MDA_PageNotFound	The page specified with *OldPageName* wasn't found.

Example

```
&ret =  RenamePage(PAGE.OrgIdTbl, PAGE.DeptIdTbl);
If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "RenamePage succeeded");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "RenamePage failed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, RenameDBField , page 368, Chapter 1, PeopleCode Built-in Functions, RenameRecord , page 371.

RenameRecord

Syntax

RenameRecord(**Record.***NewRecordName*, **Record.***OldRecordName*)

Description

Use the **RenameRecord** function to modify a record definition to have a name. This function also cleans up most references so they now use the new name.

Note. Because using this function changes records that are used to build application tables, you must rebuild (alter) the specified project before these changes can be used.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

This operation is time consuming.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
NewRecordName	Specify the new record name to be used. This name must be prefixed by the reserved word **Record**.
OldRecordName	Specify the name of the record to be changed. This name must be prefixed by the reserved word **Record**.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Bulk operation completed successfully.
%MDA_Failure	Bulk operation did not complete successfully.
%MDA_RecordNotFound	The record specified with *OldRecordName* wasn't found.

Example

```
&ret = RenameRecord(RECORD.OrgIdTbl, RECORD.DeptIdTbl);
If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "RenameRecord succeeded");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "RenameRecord failed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, RenameDBField , page 368, Chapter 1, PeopleCode Built-in Functions, RenamePage , page 370.

Repeat

Syntax

```
Repeat
  statement_list
Until logical_expression
```

Description

The **Repeat** loop causes the statements in *statement_list* to be repeated until *logical_expression* is True. Any kind of statements are allowed in the loop, including other loops. A **Break** statement inside the loop causes execution to continue with whatever follows the end of the loop. If the **Break** is in a nested loop, the **Break** does not apply to the outside loop.

Example

The following example repeats a sequence of statements until a complex Boolean condition is True:

```
Repeat
      &J = &J + 1;
      &ITEM = FetchValue(LOT_CONTROL_INV.INV_ITEM_ID, &J);
      &LOT = FetchValue(LOT_CONTROL_INV.INV_LOT_ID, &J);
Until (&ITEM = &INV_ITEM_ID And
      &LOT = &INV_LOT_ID) Or
      &J = &NUM_LOT_ROWS;
```

Replace

Syntax

Replace(*oldtext*, *start*, *num_chars*, *newtext*)

Description

Replace replaces a specified number of characters in a string.

Parameters

Parameter	Description
oldtext	A String value, part of which is to be replaced.
start	A Number designating the position in *oldtext* from which to start replacing characters.
num_chars	A Number, specifying how many characters to replace in *oldtext*.
newtext	A String value that replaces *num_chars* characters.

Returns

Returns a String value in which specific characters in *oldtext* are replaced with *newtext*.

Example

After the following statement &NEWDATESTR equals "1997":

```
&NEWDATESTR = Replace("1996",3,2,"97");
```

If this example, where the number of characters in *newtext* is less than *num_chars*, &SHORTER equals "txtx":

```
&SHORTER = Replace("txt123",4,3,"x");
```

In this example, where the number of characters in *newtext* is greater than *num_chars*, &LONGER equals "txtxxxx":

```
&LONGER = Replace("txt123",4,3,"xxxx");
```

See Also

Chapter 1, PeopleCode Built-in Functions, Substitute , page 452.

Rept

Syntax

Rept(*str*, *reps*)

Description

Rept replicates a text string a specified number of times and combines the result into a single string.

Parameters

Parameter	Description
str	A String value to be replicated.
reps	A Number value specifying how many times to replicate *str*. If *reps* is 0, **Rept** returns an empty string. If *reps* is not a whole integer, it is truncated.

Returns

Returns a String value equal to *str* repeated *reps* times.

Example

This example sets &SOMESTARS to "**********".

```
&SOMESTARS = Rept("*",10);
```

ReSubmitPubHeaderXmlDoc

Syntax

ReSubmitPubHeaderXmlDoc(*PubID, PubNode, ChannelName, VersionName*)

Description

The message header, also known as the message instance, is the published message *before* any transformations were performed. The **ReSubmitPubHeaderXmlDoc** function allows you to programmatically resubmit a message instance, as the message instance existed before any transformations were performed, much the same as you can do in the message monitor. This function resubmits the corresponding publication contract header.

You may want to use this method after an end-user has finished fixing any errors in the message data, and you want to resubmit the message, rerunning the PeopleCode.

The function is only available when the XML message has one of the following statuses:

- Error
- Timeout
- Edited
- Canceled

Parameters

Parameter	Description
PubID	Specify the PubID as a number.
PubNode	Specify the Pub node as a string.
ChannelName	Specify the channel name as a string.
VersionName	Specify the version name as a string.

Returns

A Boolean value: True if function completed successfully, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, ReSubmitPubXmlDoc , page 375.

ReSubmitPubXmlDoc

Syntax

ReSubmitPubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName, SubNode*)

Description

The **ReSubmitPubXmlDoc** function allows you to programmatically resubmit a message, much the same as you can do in the message monitor. This is the message publication as it exists *after* any transformations have been preformed. This function resubmits the corresponding publication contract.

You may want to use this method after an end-user has finished fixing any errors in the message data, and you want to resubmit the message, rerunning the PeopleCode.

The function is only available when the message has one of the following statuses:

- Error
- Timeout
- Edited
- Canceled

Parameters

Parameter	Description
PubID	Specify the PubID as a number.
PubNode	Specify the Pub node as a string.
ChannelName	Specify the channel name as a string.
VersionName	Specify the version name as a string.
MessageName	Specify the name of the message as a string.
SubNode	Specify the name of the sub node as a string.

Returns

A Boolean value: True if function completed successfully, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, ReSubmitSubXmlDoc , page 376, Chapter 1, PeopleCode Built-in Functions, ReSubmitPubHeaderXmlDoc , page 374.

ReSubmitSubXmlDoc

Syntax

ReSubmitSubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName, SubscriptionName*)

Description

The **ReSubmitSubXmlDoc** function allows you to programmatically resubmit a message, much the same as you can do in the message monitor. This function resubmits the corresponding subscription contract.

You may want to use this method after an end-user has finished fixing any errors in the message data, and you want to resubmit the message, rerunning the subscription PeopleCode.

The function is only available when the message has one of the following statuses:

- Error
- Timeout
- Edited
- Canceled

Parameters

Parameter	Description
PubID	Specify the PubID as a number.
PubNode	Specify the Pub node as a string.
ChannelName	Specify the channel name as a string.
VersionName	Specify the version name as a string.
MessageName	Specify the name of the message as a string.
SubscriptionName	Specify the name of the subscription as a string.

Returns

A Boolean value: True if function completed successfully, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, ReSubmitPubHeaderXmlDoc , page 374, Chapter 1, PeopleCode Built-in Functions, ReSubmitPubXmlDoc , page 375.

Return

Syntax

Return [*expression*]

Description

Return returns from the currently active function; the flow of execution continues from the point where the function was called. If the function returns a result, that is, if a return value is specified in the **Returns** clause of the function definition, *expression* specifies the value to pass back to the caller and must be included. If the function does not return a result, the *expression* is not allowed. If **Return** appears in a main program, it acts the same as **Exit**.

Example

In the example a Boolean return value is specified in the **Returns** clause of the **Function** statement. The **Return** statement returns a True or False value to the calling routine, based on the contents of &UPDATEOK.

```
function run_status_upd(&PROCESS_INSTANCE, &RUN_STATUS) returns Boolean;
   &UPDATEOK = SQLExec( )("update PS_PRCS_RQST set run_status = :1  where ⇒
process_instance = :2", &RUN_STATUS, &PROCESS_INSTANCE);
   If &UPDATEOK Then
      Return True;
   Else
      Return False;
   End-if;
End-function;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Function , page 196, Chapter 1, PeopleCode Built-in Functions, Exit , page 176.

ReturnToServer

Syntax

ReturnToServer({**TRUE** | **FALSE** | *&NODE_ARRAY*, | *&Message*})

Description

The **ReturnToServer** function returns a value from a PeopleCode messaging program to the publication or subscription server. You would use this in either your publication or subscription routing code, to either return an array of nodes that the message should be published to, or to do error processing (return False if entire message wasn't received.)

What is returned depends on where the PeopleCode program is called from.

From OnRoute Publication:

- TRUE: All nodes the message was published to are returned.
- FALSE: No nodes are returned (generally used with error checking).
- &NODE_ARRAY: The nodes specified in the array are returned.
- &Message: Return a response message. This must be an already instantiated message object.

Note. You can return XmlDoc objects as responses. Only homogeneous type transactions are supported, that is, you can only return an XmlDoc object as a response *if and only if* an XmlDoc object was used in the request. Similarly, you can only return a Message object if and only if a Message object was used in the request.

From OnRoute Subscription:

- TRUE: The subscription node is returned.
- FALSE: No node is returned. This is generally used with error checking.

Parameters

Parameter	Description
TRUE | **FALSE** |*&NODE_ARRAY* | *&Message*	Specify TRUE if you want publication nodes or the subscription node returned. Specify FALSE if you don't want any nodes returned, and nothing written to the database. This is generally used with error checking. Specify an object reference to an array of node names if you want to return a list of nodes to be published to. Specify a reference to a response message if you want to return a message.

Returns

None.

Example

The following is an example of a publication routing rule, which would be in the OnRoutePublication. It is used to create publication contracts.

```
local message &MSG;
local array &NODE_ARRAY;
&MSG = GetMessage();
&EMPLID = &MSG.GetRowset()(1).QA_INVEST_HDR.EMPLID.Value;
&SELECT_SQL = CreateSQL("select PUBNODE from PS_EMPLID_NODE where EMPLID = :1",⇒
 &EMPLID);
&NODE_ARRAY = CreateArray();

While &SELECT_SQL.Fetch(&PUBNODE)
   &NODE_ARRAY.Push(&PUBNODE);
End-While;
ReturnToServer(&NODE_ARRAY);
```

The following is an example of a subscription routing rule, which would be placed in the OnRouteSubscribe event:

```
local message &MSG;

&MSG = GetMessage();
&BUSINESS_UNIT = &MSG.GetRowset()(1).PO_HDR.BUSINESS_UNIT.Value;
SQLExec("Select BUSINESS_UNIT From PS_BUSINESS_UNIT where BUSINESS_UNIT = ⇒
:1",&BUSINESS_UNIT,&FOUND);
If all(&FOUND) Then
   ReturnToServer(True);
Else
   ReturnToServer(False);
End-if;
```

The following is a basic example of using an XmlDoc object:

```
Local XmlDoc &xmldoc;
. . .
/* build xmldoc */
. . .
ReturnToServer(&xmldoc);
```

See Also

Chapter 34, XmlDoc Class , page 1757, *PeopleSoft Integration Broker*, Defining Message Channels and Messages.

ReValidateNRXmlDoc

Syntax

ReValidateNRXmlDoc(*NRID*, *EntityName*)

Description

The **ReValidateNRXmlDoc** function revalidates a non-repudiation XML message. After a document has been signed and validated, you can use this function to verify it was delivered or received by the system calling the function. This function is primarily used by the Message Monitor.

Parameters

Parameter	Description
NRID	Specify the non-repudiation ID for the XML message that you want to revalidate. This parameter takes a numeric value.
EntityName	Specify the name of the entity that signed the data, as a string. For Peoplesoft, this is the node name.

Returns

A Boolean value: True if message is revalidated, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, GetNRXmlDoc , page 261, Chapter 34, XmlDoc Class , page 1757.

RevalidatePassword

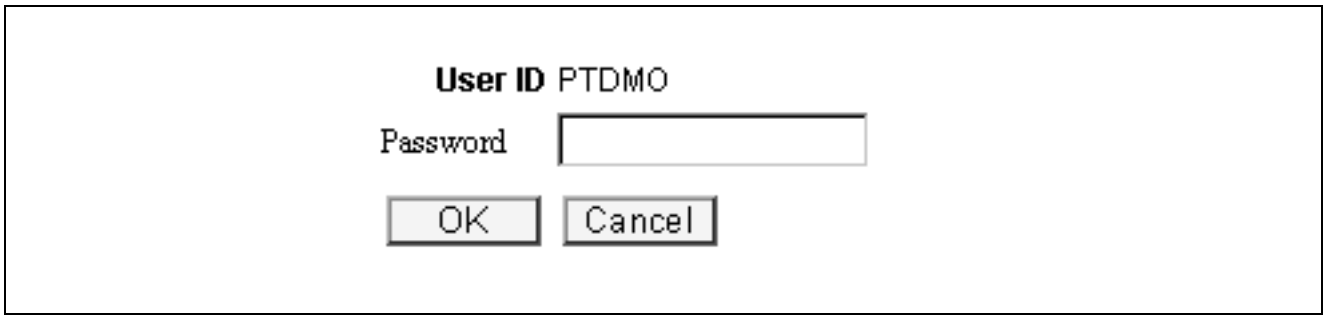
Syntax

RevalidatePassword()

Description

Use this function to validate the current end-user password.

This function displays a window prompting the user for the same password that the user signed onto the database with.



Revalidate password

Restrictions on Use in PeopleCode Events

Control does not return to the line after **RevalidatePassword** until after the user has filled in a value or pressed ENTER. This interruption of processing makes **RevalidatePassword** a “think-time” function which means that it shouldn’t be used in any of the following PeopleCode events:

- SavePreChange.
- Workflow.
- RowSelect.
- SavePostChange.
- Any PeopleCode event that fires as a result of a ScrollSelect (or one of its relatives) function calls, or a Select (or one of its relatives) Rowset class method.

See *PeopleCode Developer’s Guide*, Using Methods and Built-In Functions, Think-Time Functions.

Restrictions on Use in Signon PeopleCode

RevalidatePassword does *not* work in Signon PeopleCode. If you use this function in Signon PeopleCode, you create an infinite loop.

Returns

Returns a numeric value or a constant: you can check for either.

Value	Constant	Meaning
0	%RevalPW_Valid	Password Validated
1	%RevalPW_Failed	Password Validation Check Failed
2	%RevalPW_Cancelled	Password Validation Cancelled

Example

RevalidatePassword is commonly used in the SaveEdit PeopleCode to verify that the user entering the data is the same as the one who signed onto the database.

```
&TESTOP = RevalidatePassword();
```

```
Evaluate &TESTOP
/* Password does not match the current value */
When 1
   Error MsgGet(48, 18, "Message Not Found");
   Break;
End-Evaluate;
```

Right

Syntax

Right(*str* [, *num_chars*])

Description

Right determines a specified number of characters on the right side of a string. The function is useful if, for example, you want to get the last set of characters in a zip code or other fixed-length identification string.

Parameters

Parameter	Description
str	A String value from which you want to get the rightmost characters.
num_chars	A Number value, greater than or equal to zero. If *num_chars* is omitted it is assumed to be equal to 1.

Returns

Returns a String value equal to the rightmost *num_chars* character(s) in *str*.

Example

If &ZIP is equal to "90210-4455", the following example sets &SUFFIX to "4455":

```
&SUFFIX = Right(&ZIP, 4)
```

See Also

Chapter 1, PeopleCode Built-in Functions, Left , page 313.

Round

Syntax

Round(*dec*, *precision*)

Description

Use **Round** to round a decimal number to a specified precision.

Parameters

Parameter	Description
dec	A Number value to be rounded.
precision	A Number value specifying the decimal precision to which to round *dec*.

Returns

Returns a Number value equal to *dec* rounded up to *precision* decimal places.

Example

The following examples set the value of &TMP to 2.2, 9, then 24.09:

```
&TMP = Round(2.15,1);
&TMP = Round(8.789,0);
&TMP = Round(24.09372,2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Int , page 298, Chapter 1, PeopleCode Built-in Functions, Mod , page 327, Chapter 1, PeopleCode Built-in Functions, Truncate , page 481.

RoundCurrency

Syntax

RoundCurrency(*amt, currency_cd, effdt*)

Description

Different currencies are represented at different decimal precessions. **RoundCurrency** is a rounding function that takes currency precision into account, using a value stored in the CURRENCY_CD_TBL PeopleTools table.

Parameters

Parameter	Description
amt	The amount to be rounded.
currency_cd	The currency code.
effdt	The effective date of currency rounding.

Returns

Returns a Number value equal to *amt* rounded to the currency precision for *currency_cd*.

Example

The following example rounds 12.567 to 12.57, using the appropriate currency precision for US Dollars ("USD"):

```
&RESULT = RoundCurrency(12.567, "USD", EFFDT);
```

See Also

PeopleTools Global Technology, Controlling Currency Display Format.

RowFlush

Syntax

```
RowFlush(scrollpath, target_row)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row,
] RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

RowFlush is used to remove a specific row from a page scroll and from the component buffer.

Note. This function remains for backward compatibility only. Use the **FlushRow** Rowset method instead.

Rows that are flushed are not deleted from the database.

RowFlush is a specialized and rarely used function. In most situations, you will want to use **DeleteRow** to remove a row from the component buffer and remove it from the database as well when the component is saved.

See Also

Chapter 27, Rowset Class , FlushRow, page 1490, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number of the row to flush.

Returns

None.

Example

The following example flushes a row in a view from the component buffer:

```
RowFlush(RECORD.BNK_RCN_DTL_VW, &ROW1);
```

See Also

Chapter 1, PeopleCode Built-in Functions, ScrollFlush , page 388, Chapter 1, PeopleCode Built-in Functions, DeleteRow , page 140.

RowScrollSelect

Syntax

```
RowScrollSelect(levelnum, scrollpath,
RECORD.sel_recname
[, sqlstr [, bindvars]]
[, turbo])
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row,
] RECORD.target_recname
```

and where *bindvars* is an arbitrary-length list of bind variables in the form:

```
bindvar1 [, bindvar2]. . .
```

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

RowScrollSelect is similar to **ScrollSelect** except that it reads data from the select record into a scroll under a specific parent row, rather than automatically distributing the selected rows under the correct parent rows throughout the buffer.

Note. This function remains for backward compatibility only. Use the **Select** Rowset method instead.

You must use the WHERE clause in the SQL string to ensure that only rows that match the parent row are read into the scroll from the select record. Otherwise, all rows are read in under the specified parent row.

See Also

Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389, Chapter 27, Rowset Class , Select, page 1495, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
levelnum	Specifies the scroll level of the scroll area into which selected rows will be read. It can be 1, 2, or 3.
scrollpath	A construction that specifies a scroll level in the component buffer.
RECORD. *sel_recname*	Specifies the select record. The *selname* record must be defined in PeopleSoft Application Designer and SQL created as a table or a view, unless *sel_recname* and *target_recname* are the same. The *sel_recname* record can contain fewer fields than *target_recname*, although it must contain any key fields to maintain dependencies page records. This enables you to limit the amount of data read into the buffers on the client workstation.
sqlstr	Contains a WHERE clause to restrict the rows selected from *sel_recname* and/or an ORDER BY clause to sort the rows. The WHERE clause may contain the PeopleSoft SQL platform functions that are used for **SQLExec** processing, such as %DateIn or %Substring.
bindvars	A list of bind variables to be substituted in the WHERE clause.
turbo	Setting this Boolean parameter to True turns on Turbo ScrollSelect.

Returns

The number of rows read (optional.) This counts only lines read into the specified scroll. It does not include any additional rows read into autoselect child scrolls of the scroll.

Example

Here is an example of **RowScrollSelect** using bind variables:

```
If All(QTY_PICKED) Then
   &LEVEL1ROW = CurrentRowNumber(1);
   &LEVEL2ROW = CurrentRowNumber(2);
   &ORDER_INT_LINE_NO = FetchValue(RECORD.SHIP_SUM_INV_VW, &LEVEL1ROW, ORDER_INT_⇒
LINE_NO, &LEVEL2ROW);
   &INV_ITEM_ID = FetchValue(RECORD.SHIP_SUM_INV_VW, &LEVEL1ROW, INV_ITEM_ID,⇒
 &LEVEL2ROW);
   &QTY = RowScrollSelect(3, RECORD.SHIP_SUM_INV_VW, CurrentRowNumber(1),⇒
 RECORD.SHIP_DTL_INV_VW, CurrentRowNumber(2), RECORD.DEMAND_LOC_INV, RECORD.DEMAND_⇒
LOC_INV, "WHERE BUSINESS_UNIT = :1 AND ORDER_NO = :2 AND DEMAND_SOURCE = :3 AND⇒
 SOURCE_BUS_UNIT = :4 AND ORDER_INT_LINE_NO = :5 AND SCHED_LINE_NO = :6 AND INV_⇒
ITEM_ID = :7 AND DEMAND_LINE_NO = :8", SHIP_HDR_INV.BUSINESS_UNIT, ORDER_NO,⇒
 DEMAND_SOURCE, SOURCE_BUS_UNIT, ORDER_INT_LINE_NO, SCHED_LINE_NO, INV_ITEM_ID,⇒
 DEMAND_LINE_NO, True);
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, RowScrollSelectNew , page 386, Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389, Chapter 1, PeopleCode Built-in Functions, ScrollSelectNew , page 392, Chapter 1, PeopleCode Built-in Functions, ScrollFlush , page 388, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442.

RowScrollSelectNew

Syntax

```
RowScrollSelectNew(levelnum, scrollpath,
RECORD.sel_recname,
[sqlstr [, bindvars]]
[, turbo])
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row,
] RECORD.target_recname
```

where *bindvars* is an arbitrary-length list of bind variables in the form:

```
binvar1 [, bindvar2]. . .
```

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

RowScrollSelectNew is similar to **RowScrollSelect**, except that all rows read into the work scroll are marked *new* so they are automatically inserted into the database at Save time.

Note. This function remains for backward compatibility only. Use the **SelectNew** Rowset method instead.

This capability can be used, for example, to insert new rows into the database by selecting data using a view of columns from another database tables.

See Also

Chapter 1, PeopleCode Built-in Functions, RowScrollSelect , page 384, Chapter 27, Rowset Class , SelectNew, page 1498, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
level	Specifies the scroll level of the scroll area into which selected rows are read. It can be 1, 2, or 3.
scrollpath	A construction that specifies a scroll level in the component buffer.
RECORD. *sel_recname*	Specifies the select record. The *selname* record must be defined in the record definition and SQL created as a table or a view, unless *sel_recname* and *target_recname* are the same. The *sel_recname* record can contain fewer fields than *target_recname*, although it must contain any key fields to maintain dependencies with other page records. This enables you to limit the amount of data read into the buffers on the client workstation.
sqlstr	Contains a WHERE clause to restrict the rows selected from *sel_recname* and/or an ORDER BY clause to sort the rows. The WHERE clause may contain the PeopleSoft SQL platform functions that are used for **SQLExec** processing, such as %DateIn or %Substring.
bindvars	A list of bind variables to be substituted in the WHERE clause. The same restrictions that exist for **SQLExec** exist for these variables.
turbo	Setting this Boolean parameter to True turns on Turbo ScrollSelect. This will improve the performance of ScrollSelect verbs by as much as 300%, but should be implemented with caution on existing applications.

Returns

The number of rows read (optional.) This counts only lines read into the specified scroll. It does not include any additional rows read into autoselect child scrolls of the scroll.

Example

The following example reads rows into the level 2 scroll and marks the rows as new:

```
&QTY = RowScrollSelectNew(2, RECORD.BI_LINE_VW, &ROW1, RECORD.BI_LINE_DST,⇒
 RECORD.BI_LINE_DST, "where business_unit = :1 and invoice = :2 and line_seq_num =⇒
 :3", BI_HDR.BUSINESS_UNIT, BI_HDR.INVOICE, &CURR_LINE_SEQ);
```

See Also

Chapter 1, PeopleCode Built-in Functions, RowScrollSelect , page 384, Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389, Chapter 1, PeopleCode Built-in Functions, ScrollSelectNew , page 392, Chapter 1, PeopleCode Built-in Functions, ScrollFlush , page 388, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442.

RTrim

Syntax

RTrim(*source_str* [, *trim_str*])

Description

Use the **RTrim** function to remove characters, usually trailing blanks, from the right of a string.

Parameters

Parameter	Description
source_str	A String from which you want to remove trailing characters.
trim_str	A String consisting of a list of characters, all occurrences of which are removed from the right of *source_str*. Characters in *trim_str* that occur in *source_str* to the left of any character not in *trim_str* are be removed. If this parameter is not specified, " " is assumed.

Returns

Returns a String formed by deleting, from the end of *source_str*, all occurrences of each character specified in *trim_str*.

Example

The following example removes trailing blanks from &NAME and places the results in &TRIMMED:

```
&TRIMMED = RTrim(&NAME);
```

The following example removes trailing punctuation marks from REC.INP and places the results in &TRIMMED:

```
&TRIMMED = RTrim(REC.INP, ".,;:!?");
```

See Also

Chapter 1, PeopleCode Built-in Functions, LTrim , page 317.

ScheduleProcess

Syntax

```
ScheduleProcess(process_type, process_name
[, run_location] [, run_cntl_id] [, process_instance]
[, run_dttm] [, recurrence_name] [, server_name])
```

Description

ScheduleProcess schedules a process or job, writing a row of data to the Process Request table (PSPRCSRQST).

Note. This function is no longer supported. Use the **CreateProcessRequest** function instead.

See Also

Chapter 1, PeopleCode Built-in Functions, CreateProcessRequest , page 104.

ScrollFlush

Syntax

```
ScrollFlush(scrollpath)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_row, ]
```

RECORD.*target_recname*

To prevent ambiguous references, you can use **SCROLL.** *scrollname,* where *scrollname* is the same as the scroll level's primary record name.

Description

ScrollFlush removes all rows inside the target scroll area and frees its associated buffer.

Note. This function remains for backward compatibility only. Use the **Flush** Rowset method instead.

Rows that are flushed are not deleted from the database. This function is often used to clear a work scroll before a call to ScrollSelect.

See Also

Chapter 27, Rowset Class , Flush, page 1489, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.

Returns

None.

Example

The following example clears the level-one scroll then fills the level-one and level-two scrolls.

```
/* Throw away all rows */
ScrollFlush(RECORD.DBFIELD_VW);
/* Fill in new values */
&FIELDSEL = "where FIELDNAME like '" | FIELDNAME | "%'";
&ORDERBY = " order by FIELDNAME";
ScrollSelect(1, RECORD.DBFIELD_VW, RECORD.DBFIELD_VW, &FIELDSEL | &ORDERBY);
ScrollSelect(2, RECORD.DBFIELD_VW, RECORD.DBFIELD_LANG_VW, RECORD.DBFIELD_LANG_VW,⇒
 &FIELDSEL | " and LANGUAGE_CD = :1" | &ORDERBY, DBFIELD_SRCH.LANGUAGE_CD);
```

See Also

Chapter 1, PeopleCode Built-in Functions, RowFlush , page 383, Chapter 1, PeopleCode Built-in Functions, RowScrollSelect , page 384, Chapter 1, PeopleCode Built-in Functions, RowScrollSelectNew , page 386, Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389, Chapter 1, PeopleCode Built-in Functions, ScrollSelectNew , page 392.

ScrollSelect

Syntax

ScrollSelect(levelnum,

```
[RECORD.level1_recname, [RECORD.level2_recname,]] RECORD.target_recname, RECORD.sel_recname
   [, sqlstr [, bindvars]]
   [, turbo])
```

where *bindvars* is an arbitrary-length list of bind variables in the form:
bindvar1 [, *bindvar2*]. . .

Description

ScrollSelect, like the related ScrollSelect functions (**ScrollSelectNew**, **RowScrollSelect**, and **RowScrollSelectNew**) reads data from database tables or views into a scroll area on the active page.

Note. This function remains for backward compatibility only. Use the **Select** Rowset method instead.

See Chapter 27, Rowset Class , Select, page 1495.

Using ScrollSelect

ScrollSelect automatically places child rows in the target scroll area under the correct parent row in the next highest scroll area. If it cannot match a child row to a parent row an error occurs.

ScrollSelect selects rows from a table or view and adds the rows to a scroll area of a page. Let's call the record definition of the table or view that it selects from the *select record*; and let's call the record definition normally referenced by the scroll area (as specified in the page definition) the *default scroll record*. The select record can be the same as the default scroll record, or it can be a different record definition that has compatible fields with the default scroll record. If you define a select record that differs from the default scroll record, you can restrict the number of fields that are loaded into the buffers on the client workstation by including only the fields that you actually need.

ScrollSelect accepts an SQL string that can contain a WHERE clause restricting the number of rows selected into the scroll area. The SQL string can also contain an ORDER BY clause to sort the rows.

ScrollSelect functions generate a SQL SELECT statement at runtime, based on the fields in the select record and WHERE clause passed to them in the function call. This gives ScrollSelect functions a significant advantage over **SQLExec**: they enable you to change the structure of the select record without affecting the PeopleCode, unless the field affected is referred to in the WHERE clause string. This can make the application easier to maintain.

Often, ScrollSelect is used to select rows into a *work scroll*, which is sometimes hidden using **HideScroll**. A work scroll is a scroll in which the **No Auto Select** option is selected on the record definition in PeopleSoft Application Designer so that PeopleTools does not automatically populate the scroll at page startup. You can right-click on the scroll or grid then select the scroll's **No Auto Select** attribute check box in the record property dialog box.

Depending on how you intend the scroll to be used by the end-user, you may also want to select **No Auto Update** to prevent database updates, and prevent row insertions or deletions in the scroll area by selecting **No Row Insert** or **No Row Update.**

To call **ScrollSelect** at page startup, place the function call in the RowInit event of a key field on the parent scroll record. For example, if you want to fill scroll level one, place the call to **ScrollSelect** in the RowInit event of a field on level zero.

Parameters

Parameter	Description
levelnum	Specifies the level of the scroll level to be filled (the target scroll area. The value can be 1, 2, or 3.
target_recname	Specifies a record identifying the target scroll, into which the selected rows are read. If *target_recname* is on scroll level 2, it must be proceeded by a **RECORD.** *level1_recname*. If it is on level 3, you must specify both **RECORD.** *level1_recname* and **RECORD.** *level2_recname*.
RECORD. *sel_recname*	Specifies the select record. The *selname* record must be defined in PeopleSoft Application Designer and SQL created (using **Build**, **Project**) as a table or a view, unless *sel_recname* and *target_recname* are the same. The *sel_recname* record can contain fewer fields *target_recname*, although it must contain any key fields to maintain dependencies with other page records. This enables you to limit the amount of data read into the buffers on the client workstation.
sqlstr	Contains a WHERE clause to restrict the rows selected from *sel_recname* and/or an ORDER BY clause to sort the rows. The WHERE clause can contain the PeopleSoft meta-SQL functions such as %Datein or %CurrentDateIn. It can also contain inline bind variables.
bindvars	A list of bind variables to be substituted in the WHERE clause. The same restrictions that exist for **SQLExec** exist for these variables.
turbo	Setting this Boolean parameter to True turns on Turbo ScrollSelect. This improves the performance of ScrollSelect verbs by as much as 300%, but should be implemented with caution on existing applications, which may depend on the behavior of **ScrollSelect** without Turbo.

Returns

The number of rows read (optional.) This counts only lines read into the specified scroll. It does not include any additional rows read into autoselect child scrolls of the scroll.

Example

This example uses WHERE clauses to reset the rows in two scroll areas:

```
&FIELD_CNT = ActiveRowCount(DBFIELD_VW.FIELDNAME);
For &I = 1 to &FIELD_CNT;
   If RecordChanged(DBFIELD_VW.FIELDNAME, &I, DBFIELD_LANG_VW.FIELDNAME, 1)    Then
      &FIELDNAME = FetchValue(DBFIELD_VW.FIELDNAME, &I);
      &RET = WinMessage("Descriptions for " | &FIELDNAME | " have been changed. ⇒
 Discard changes?", 289, "DBField Changed!");
      If &RET = 2 Then
         /* Cancel selected */
         Exit;
      End-if;
   End-if;
End-for;
/* Now throw away all rows */
ScrollFlush(RECORD.DBFIELD_VW);
/* Fill in new values */
```

```
&FIELDSEL = "where FIELDNAME like '" | FIELDNAME | "%'";
&ORDERBY = " order by FIELDNAME";
&QTY1 = ScrollSelect(1, RECORD.DBFIELD_VW, RECORD.DBFIELD_VW, &FIELDSEL |⇒
 &ORDERBY);
&QTY2 = ScrollSelect(2, RECORD.DBFIELD_VW, RECORD.DBFIELD_LANG_VW, RECORD.DBFIELD_⇒
LANG_VW, &FIELDSEL | " and LANGUAGE_CD = :1" | &ORDERBY, ⇒
DBFIELD_SRCH.LANGUAGE_CD);
```

See Also

Chapter 1, PeopleCode Built-in Functions, RowScrollSelect , page 384, Chapter 1, PeopleCode Built-in Functions, RowScrollSelectNew , page 386, Chapter 1, PeopleCode Built-in Functions, ScrollFlush , page 388, Chapter 1, PeopleCode Built-in Functions, ScrollSelectNew , page 392, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442, *PeopleCode Developer's Guide*, Data Buffer Access.

ScrollSelectNew

Syntax

```
ScrollSelectNew(levelnum,
   [RECORD.level1_recname,
   [RECORD.level2_recname, ]]
   RECORD.target_recname,
   RECORD.sel_recname [, sqlstr [, bindvars]]
   [, turbo])
```

and where *bindvars* is an arbitrary-length list of bind variables in the form:
bindvar1 [, *bindvar2*]. . .

Description

ScrollSelectNew is similar to **ScrollSelect**, except that all rows read into the work scroll are marked *new* so they are automatically inserted into the database at Save time.

Note. This function remains for backward compatibility only. Use the **SelectNew** Rowset class method instead.

This capability can be used, for example, to insert new rows into the database by selecting data using a view of columns from other database tables.

See Also

Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389, Chapter 27, Rowset Class , SelectNew, page 1498, *PeopleCode Developer's Guide*, Data Buffer Access.

Parameters

Parameter	Description
levelnum	Specifies the level of the scroll level to be filled (the target scroll area. The value can be 1, 2, or 3.
target_recname	Specifies a record identifying the target scroll, into which the selected rows are read. If *target_recname* is on scroll level 2, it must be proceeded by a **RECORD.** *level1_recname*. If it is on level 3, you must specify both **RECORD.** *level1_recname* and **RECORD.** *level2_recname*.
RECORD. *sel_recname*	Specifies the select record. The *selname* record must be defined in PeopleSoft Application Designer and SQL created as a table or a view (using **Build**, **Project**), unless *sel_recname* and *target_recname* are the same. The *sel_recname* record can contain fewer fields *target_recname*, although it must contain any key fields to maintain dependencies with other records on the page. This enables you to limit the amount of data read into the buffers on the client workstation.
sqlstr	Contains a WHERE clause to restrict the rows selected from *sel_recname* and/or an ORDER BY clause to sort the rows. The WHERE clause may contain the PeopleSoft SQL platform functions that are used for **SQLExec** processing, such as %Datein or %Substring.
bindvars	A list of bind variables to be substituted in the WHERE clause. The same restrictions that exist for **SQLExec** exist for these variables.
turbo	Setting this Boolean parameter to True turns on Turbo ScrollSelect. This improves the performance of ScrollSelect verbs by as much as 300%, but should be implemented with caution on existing applications.

Returns

The number of rows read (optional.) This counts only lines read into the specified scroll. It does not include any additional rows read into autoselect child scrolls of the scroll.

Example

The following statement selects rows from DATA2 and reads them into scroll level one on the page. If the end-user saves the page, these rows will be inserted into DATA1:

```
&QTY = ScrollSelectNew(1, RECORD.DATA1, RECORD.DATA2,
   "Where SETID = :1 and CUST_ID = :2",
   CUSTOMER.SETID, CUSTOMER.CUST_ID);
```

See Also

Chapter 1, PeopleCode Built-in Functions, RowScrollSelect , page 384, Chapter 1, PeopleCode Built-in Functions, RowScrollSelectNew , page 386, Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389, Chapter 1, PeopleCode Built-in Functions, ScrollFlush , page 388, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442.

Second

Syntax

Second(*timevalue*)

Description

Use the **Second** function to extract the seconds component of a Time value.

Parameters

Parameter	Description
timevalue	A Time value from which to extract seconds.

Returns

Returns a Number equal to the seconds part of *timevalue*.

Example

Assume that &TIMEOUT contains Time value of 16:48:21. The following would set &SECS to 21:

```
&SECS = Second(&TIMEOUT);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Hour , page 291, Chapter 1, PeopleCode Built-in Functions, Minute , page 326.

SendMail

Syntax

```
SendMail(flags, recipients, CCs, BCCs, subject, text,
         [, attachment_filenames][, attachment_titles] [, Mail_From] [,mail_sep] [, Content_Type]
```

Description

Use the **SendMail** function to send an email message from a PeopleSoft page. The **SendMail** function supports VIM and MAPI email subsystems for two-tier architecture, and SMTP for three-tier. The APIs that support these subsystems must be present on the system for the function to work.

Note. The code that actually calls **SendMail** determines the run location (two-tier or three-tier) *not* the originating code. For example, if an online page (two-tier) calls a Component Interface (three-tier) that calls SendMail, the program is running in three-tier, not two-tier mode. The only exception to this is if SendMail is used in a program running on the batch server (for example, if an Application Engine PeopleCode program used SendMail.) In a batch server environment, SendMail *always* uses SMTP.

The function sends a message using standard mail options, including recipient, CC, BCC, subject, and the text of the note. The message can include attached files, for which you supply fully qualified file names (that is, file names with paths) and titles (which appear in place of the fully qualified filename in the message). In addition, you can specify a *sender*, to be used as the 'reply-to' field. You can also use the *Reply_To* parameter to specify the value to be used by the receiver for replying.

The *flags* parameter of this function and its return codes are platform dependent: the values are different depending on whether the platform is using the VIM or MAPI mail API (the SMTP mail API ignores any additional flags.) For this reason, **SendMail** should be used only in customizations that run on a known platform, if it is used at all. The preferred, platform-independent method for sending email from PeopleCode is to define an email routing as part of a business event, then trigger the business event using **TriggerBusinessEvent**.

For workflow, in the activity email routing, you can specify the field map for *Reply_To* and *Sender*.

See Also

Chapter 1, PeopleCode Built-in Functions, TriggerBusinessEvent , page 481.

Parameters

Parameter	Description
flags	An integer value passed directly to the mail system API to control mail system options. The value passed in this parameter is platform-dependent. The SMTP mail API ignores this parameter. The following values can be used with MAPI. You can combine different options by adding any of the following values and passing the sum. Pass zero to turn all of the options off.

Value	Description
1	Allows a logon interface if required.
2	Prevents Simple MAPI from using an existing shared session if one is present.
8	Displays a dialog box which enables the user to create or modify the message.

Parameter	Description
recipients	A string consisting of a semicolon-separated list of email addresses containing the names of the message's primary recipients.
CCs	A string consisting of a semicolon-separated list of email addresses that are sent copies of the message.
BCCs	A string consisting of a semicolon-separated list of email addresses that are sent copies of the message. These recipients won't appear on the message list.
subject	A string containing the text that appears in the message's Subject field.
text	The text of the message.
attachment_filenames	A string consisting of a semicolon-separated list of fully qualified filenames, containing the complete path to the file and the filename itself.
attachment_titles	Another semicolon-separated list containing titles for each of the files provided in the *attachment_filenames* parameter. The titles appear near the attachment icons in place of the fully qualified filename.
Mail_From	A string used to populate the 'reply-to' field. If this field isn't used, the sender address from application server config file is used.
Mail_Sep	Specify the delimiter to be used to separate one email address from another. The default value is a semicolon (;).
Content_Type	Specify the content type of the email as a string. The default value is plain text. If you want to specify HTML, you should use the following: `Content-type: text/html; charset=US-ASCII`
Reply_To	Specify the email address that the receiver should use when replying to this email.
Sender	Specifies who the email is from, as a string. This may be different than the values specified for *Mail_From* or *Reply_To* parameters.

Returns

Returns a Number value, which, if not one of the following general return codes, is platform-dependent.

General Return Codes

Return Code	Description
0	No Error
-1	No mail interface installed.

VIM Return Codes

Return Code	Description
0	SMISTS_SUCCESS
1	SMISTS_FAILURE
4	SMISTS_ATTACHMENT_NOT_FOUND
8	SMISTS_INSUFFICIENT_MEMORY
15	SMISTS_NAME_NOT_FOUND
16	SMISTS_NOT_SUPPORTED
22	SMISTS_OPEN_FAILURE
128	SMISTS_INVALID_ADDR_BOOK
129	SMISTS_TOO_MANY_FILES
130	SMISTS_TOO_MANY_RECIPIENTS
131	SMISTS_USER_CANCEL

MAPI Return Codes

Return Code	Description
1	MAPI_USER_ABORT
2	MAPI_E_FAILURE
3	MAPI_E_LOGIN_FAILURE
4	MAPI_E_DISK_FULL
5	MAPI_E_INSUFFICIENT_MEMORY
6	MAPI_E_ACCESS_DENIED

Return Code	Description
8	MAPI_E_TOO_MANY_SESSIONS
9	MAPI_E_TOO_MANY_FILES
10	MAPI_E_TOO_MANY_RECIPIENTS
11	MAPI_E_ATTACHMENT_NOT_FOUND
12	MAPI_E_ATTACHMENT_OPEN_FAILURE
13	MAPI_E_ATTACHMENT_WRITE_FAILURE
14	MAPI_E_UNKNOWN_RECIPIENT
15	MAPI_E_BAD_RECIPTYPE
16	MAPI_E_NO_MESSAGES
17	MAPI_E_INVALID_MESSAGE
18	MAPI_E_TEXT_TOO_LARGE
19	MAPI_E_INVALID_SESSION
20	MAPI_E_TYPE_NOT_SUPPORTED
21	MAPI_E_AMBIGUOUS_RECIPIENT
22	MAPI_E_MESSAGE_IN_USE
23	MAPI_E_NETWORK_FAILURE
24	MAPI_E_INVALID_EDITFIELDS
25	MAPI_E_INVALID_RECIPS
26	MAPI_E_NOT_SUPPORTED

Note. Additional VIM and MAPI error codes may be generated, depending on your email provider.

There are no special return codes for SMTP.

Example

The following example sets up several variables that are then used to construct an email message that includes two attachments:

```
&MAIL_FLAGS = 0;
&MAIL_TO = "dduffield@peoplesoft.com;sweet_pea@peoplesoft.com";
&MAIL_CC = "";
&MAIL_BCC = "mom@aol.com";
&MAIL_SUBJECT = "Live long and prosper!";
&MAIL_TEXT = "Please read my attached CV.  You will be amazed and hire me⇒
 forthwith.";
&MAIL_FILES = "c:\mydocs\resume.doc;c:\mydocs\coverlet.doc";
&MAIL_TITLES = "My CV;READ ME";
&MAIL_SENDER = "MyEmail@Yahoo.com";
&RET = SendMail(&MAIL_FLAGS, &MAIL_TO, &MAIL_CC, &MAIL_BCC, &MAIL_SUBJECT, ⇒
&MAIL_TEXT, &MAIL_FILES, &MAIL_TITLES, &MAIL_SENDER);
if not (&RET = 0) then
   WinMessage("Return status from mail = " | &RET);
end-if;
```

SetAuthenticationResult

Syntax

SetAuthenticationResult(*AuthResult* [, *UserId*] [,*ResultDocument*] [,*PasswordExpired*] [*DaysLeftBeforeExpire*])

Description

The **SetAuthenticationResult** function is used in signon PeopleCode to customize the authentication process. It enables the developer using Signon PeopleCode to implement additional authentication mechanisms beyond the basic PeopleSoft ID and Password authentication.

When *PasswordExpired* is True, it indicates the password is expired, the passwordexpired.html page is displayed during login when signon PeopleCode is enabled.

When *DaysLeftBeforeExpire* is greater than 0, and *PasswordExpired* is False, indicating that the password will expire in x days, the passwordwarning.html page is displayed during login when signon PeopleCode is enabled.

Note. If you set *AuthResult* to False, *ResultDocument* must be the text of an error message. This text is displayed on the signon screen.

Parameters

Parameter	Description
AuthResult	Specify whether the authentication is successful. This parameter takes a Boolean value. If True is used, the end-user of the UserId specified on the Signon page is allowed access to the system. When AuthResult is True, the customer is responsible for providing a logout to the end user. They will remain logged in until a logout command is issued from the user, or the session expires.
UserId	Specify the UserId of the user signing on. The default value is the UserId entered on the signon page. This parameter takes a string value. This is the value returned by %SignonUserId
ResultDocument	When *ResultDocument* is blank (""), this parameter value is ignored. Otherwise, specify a message to be displayed in the signonresultdoc.html file when *AuthResult* is True. If AuthResult is False, the ResultDocument text value is displayed on the signon screen. If ResultDocument has a value, any values in *PasswordExpired* and *DaysLeftBeforeExpire* are ignored.
PasswordExpired	Specify if the user's password has expired. The values are: • False (default) if the user's password hasn't expired. • True if the user's password has expired If this value is specified as True, the user is allowed to log in, but is able to access only a limited portion of the system: just enough to change their expired password.
DaysLeftBeforeExpire	A numeric value indicating the number of days left before the password expires. If the value is greater than 0, a warning is displayed when Authorized is True and Expired is False.

Returns

A Boolean value: True if function completed successfully, False otherwise.

Example

```
If updateUserProfile(%SignonUserId, %SignonUserPswd, &array_attribs) Then
   SetAuthenticationResult(True, &SignonUserID, "", False);
End-If;
```

The following example is within a function used for logging onto a system:

```
   If (AddToDateTime(&fmc_wsl_exp_date, 0, 0, 0, 0, 10, 0) >= %Datetime) Then
         /* WSL logon was within last x minutes, so accept WSL for PS logon */
         SetAuthenticationResult( True, Upper(&userID), "", False);
      Else
         /* WSL logonn was too long ago, so request a more recent WSL logon */
         SetAuthenticationResult( False, "getmorerecentcookie", "", False,7); ⇒
/*displays the customized passwordwarning.html. */
      End-If;
```

In the following example, AuthResult is True and ResultDocument is set as text to be displayed in an HTML tag.

```
SetAuthenticationResult( True, &USERID, "Result Doc Text", False, 0);
```

As part of this example, specify the following in the configuration properties:

```
singonresultdoc_page=signonresultdoctext.html
```

In signonresultdoctext.html, add a meta field as follows:

```
<%=resultDoc%>:

<html>
....
  <tr><td class="PSSRCHACTION" no wrap=true><%=resultDoc%></td></tr>
.....
</html>
```

This produces a screen shot similar to the following:

Example with Result Doc Text

See Also

Chapter 3, System Variables , %ResultDocument , page 576, Chapter 3, System Variables , %AuthenticationToken , page 564, *PeopleTools Security*, Understanding PeopleSoft Security, PeopleSoft Signon.

SetChannelStatus

Syntax

SetChannelStatus(*ChannelName*, *Status*)

Description

The **SetChannelStatus** enables you set the status of the specified channel. You could use this function to restart a channel that had been paused, or pause a running channel.

Parameters

Parameter	Description
ChannelName	Specify the channel name.
Status	Specify the status you want to set the channel to. The values are: • 1 for Run • 2 for Pause

Returns

A Boolean value: True if the channel status was changed successfully. False otherwise.

Example

```
/* User has clicked on a channel to change its status */

If CHNL_STATUS = "1" Then
   rem running, so pause;
   &status = 2;
Else
   rem paused. So run;
   &status = 1;
End-If;

If SetChannelStatus(AMM_CHNL_SECVW.CHNLNAME, &status) Then
   CHNL_STATUS = String(&status);
Else
   MessageBox(0, MsgGetText(117, 1, ""), 117, 22, "");
End-If;
```

See Also

Chapter 17, Message Class , page 973.
PeopleTools PeopleBook: Peoplesoft Integration Broker

SetControlValue

Syntax

SetControlValue(*Value*, *PageName*, *PageFieldName* [, *RowNumber*] [, *&Field*])

Description

The **SetControlValue** function enables you to set an override string on the current field so that it simulates an end-user entering data.

When a page is refreshed after a PeopleCode program completes, each field value gets set from the buffer. However, if you use this function to specify an override string for a field, the value you specify is used *instead* of the value in the buffer. This value is inserted directly into the control on the page, as if the end-user typed it in. The field buffer remains unchanged. All validations, FieldEdit and FieldChange PeopleCode run immediately.

This function can be used in the following scenario: Suppose you have a text field that has a menu pop-up associated with it. The end-user can use a secondary page to select an item to be used for the value. From the menu PeopleCode, you can verify that the value is valid, but the field doesn't turn red and the end-user can leave the field. This could potential mean saving the page with bad data. You can use this function after the secondary page is dismissed. This causes the same edits to be run as if the end-user had typed in the value.

This function doesn't work for radio button or check box controls.

Considerations With Field Verification

SetControlValue only sets the value of the field. If you specify an incorrect value, SetControlValue has an error at runtime.

For example, suppose you are setting a value like "1900-01-01" into a date field that is expecting the format 01/01/1900. If the end-user entered 1900-01-01 they would get an error, so SetControlValue causes an error with this value also. You may want to use a value in the format the end-user might enter. You can get this value by using the FormattedValue method on a field. For example:

```
&DATE_IN_EFFECT = SF_PRDN_AREA_IT.DATE_IN_EFFECT.FormattedValue;
...
SetControlValue(&DATE_IN_EFFECT, %Page, "DATE_IN_EFFECT", &OCCURSNUM);
```

The FormattedValue function converts the field value from the PeopleSoft representation to the representation the end-user would see and enter.

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Parameters

Parameter	Description
Value	Specify an override value on the current field. This parameter takes a string value.
pagename	Specify the name of page where the field exists.
pagefieldname	Specify the page field name. This is *not* the name of the field. This is the name that is assigned to the field in PeopleSoft Application Designer, on the page field properties.
RowNumber	Specify the row number of the field. The default value is 1 if this parameter isn't set.
&Field	Specify an already instantiated field object referencing the field you want to override.

Note. If you want to set an override string for a field on the level 1 scroll for a page, you don't need to specify either a row number or a field object. However, if you want to set the override string for a field on either the second or third level scroll for a page, you must specify *both* a row number and a field object for SetControlValue to work.

Returns

None.

Example

```
Declare Function item_seach PeopleCode FUNCLIB_ITEM.INV_ITEM_ID FieldFormula;

&SEARCHREC = "PS_" | RECORD.MG_ITEM_OWN1_VW;
item_seach("", SF_PRDN_AREA.BUSINESS_UNIT, "ITEM", &SEARCHREC, "", &INV_ITEM_ID,⇒
 "");
SetControlValue(&INV_ITEM_ID);
```

The following example is used in the PeopleSoft Internet Architecture:

```
Declare Function item_search PeopleCode FUNCLIB_ITEM.INV_ITEM_ID FieldFormula;

Component string &ITEM_ID_SEARCH;

&ITEMRECNAME = "PS_" | Record.MG_ITEM_PDO_VW;
item_serach("", EN_PDO_WRK.BUSINESS_UNIT, "ITEM", &ITEMRECNAME, "", &INV_ITEM_ID,⇒
 "");
If All(&INV_ITEM_ID) Then
   Evaluate &ITEM_ID_SEARCH
   When "F"
      SetControlValue(&INV_ITEM_ID, Page.EN_PDO_COPY, "FROM_ITEMID")
   When "T"
      SetControlValue(&INV_ITEM_ID, Page.EN_PDO_COPY, "TO_ITEMID")
   End-Evaluate;
End-If;
```

SetCursorPos

Syntax

SetCursorPos(**Page**.*pagename*, *scrollpath*, *target_row*, [*recordname*.]*fieldname*)

where *scrollpath* is:
[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD**.*target_recname*

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

SetCursorPos places the focus in a specific field anywhere in the current component. To transfer to a page outside the current component, use **Transfer**.

You can use the **SetCursorPos** function in combination with an **Error** or **Warning** in SaveEdit to place the focus on the field that caused the error or warning condition. You must call **SetCursorPos** *before* an **Error** statement, because **Error** in SaveEdit terminates all save processing, including the program from which it was called.

See Also

Chapter 1, PeopleCode Built-in Functions, Transfer , page 463, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
Pagename	The name of the page specified in the page definition, preceded by the keyword **Page**. The *pagename* page must be in the current component. You can also pass the %page system variable in this parameter (without the **Page** reserved word).
scrollpath	A construction that specifies a scroll level in the component buffer.
[*recordname* **.**]*fieldname*	Specify a field designating the record and field in the scroll where you want to place the cursor.
target_row	The row number of the row in which you want to place the cursor.

Returns

None.

Example

The following example places the cursor in the appropriate field if a SaveEdit validation fails. Note the use of the %page system variable to get the page name. Note also that **SetCursorPos** is called before **Error**.

```
If None(&ITEM_FOUND) Then
      SetCursorPos(%Page, INV_ITEM_ID, CurrentRowNumber());
      Error (MsgGet(11100, 162, "Item is not valid in the order business unit.",⇒
 INV_ITEM_ID, CART_ATTRIB_INV.ORDER_BU));
End-If;
```

The following example is similar, but uses the **Page** reserved word and page name:

```
If %Component = COMPONENT.BUS_UNIT_TBL_GL Then
      SetCursorPos(PAGE.BUS_UNIT_TBL_GL1, DEFAULT_SETID, CurrentRowNumber());
End-If;
Error MsgGet(9000, 165, "Default TableSet ID is a required field.");
```

See Also

Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475.

SetDBFieldAuxFlag

Syntax

SetDBFieldAuxFlag(Field.*FieldName***,** *FlagNumber***,** *Setting***)**

Description

Use the **SetDBFieldAuxFlag** function to set the auxiliary flag mask (AuxFlagMask) property for the specified field. This field indicates properties about the field.

Currently, only one flag comes preset from PeopleSoft: a 1 indicates a ChartField. If you want to associate a property with a field, you must coordinate with other developers to make certain that no one else is setting a property using the same flag number.

Use the GetAuxFlag Field method to read the current setting of the property.

If you use this function, the change is made to the database field, but it doesn't require a rebuild of the database. However, the change is *not* reflected in the component buffer. You must reload the component for the new setting to take place.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
Fieldname	Specify the name of the field that you want the AuxMaskFlag property changed. This name must be prefixed by the reserved word **Field**.
FlagNumber	Specify the flag value, a number between 1 and 32. A 1 is a ChartField.
Setting	Specify whether this flag should be on (True) or off (False).

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Bulk operation completed successfully.
%MDA_Failure	Bulk operation did not complete successfully.
%MDA_FieldNotFound	The specified field was not found in the database.

Example

```
&ret = SetDBFieldAuxFlag(Field.OrgId, 1, True);
If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldAuxFlag succeeded");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldAuxFlag failed");
End-If;
```

See Also

Chapter 12, Field Class , GetAuxFlag, page 846.

SetDBFieldCharDefn

Syntax

```
SetDBFieldCharDefn(Field.FieldName, Length [, FormatFamily])
```

Description

Use the SetDBFieldCharDefn function to create a field definition of type character, with the indicated name, length, and format family.

Note. After using this function, you should use the **SetDBFieldLabel** function to define the label for the new field.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
Fieldname	Specify the name of the new field that you want to create. This name must be prefixed by the reserved word **Field**.
Length	Specify the length of the new field as a number.
FormatFamily	Specify the format family of the new field. This parameter is optional: the default value is upper case. The valid values are: • %FormatFamilyType_Upper (default) • %FormatFamilyType_Name • %FormatFamilyType_Phone • %FormatFamilyType_Zip • %FormatFamilyType_SSN • %FormatFamilyType_MixedCase • %FormatFamilyType_NumOnly • %FormatFamilyType_SIN • %FormatFamilyType_PhoneIntl • %FormatFamilyType_ZipIntl

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Bulk operation completed successfully.
%MDA_Failure	Bulk operation did not complete successfully.
%MDA_Duplicate	The field specified by *FieldName* already exists.
%MDA_FieldFmtLength	The specified length conflicts with the specified format family and was overwritten when the field was created.

Example

```
&ret = SetDBFieldCharDefn(Field.OrgId, 10,
%FormatFamilyType_MixedCase);
If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldCharDefn succeeded");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldCharDefn failed");
End-If;
```

You can also use this function with de-referenced parameters, as follows:

```
&ret = SetDBFieldCharDefn(@("FIELD." | FS_CF_UPD_AET.FIELDNAME), FS_CF_UPD_AET.NEW_⇒
CF_LENGTH, %FormatFamilyType_MixedCase);
```

The following example adds a new character field:

```
&cf = "CF1";
&len = 10;
&frmt = %FormatFamilyType_Upper;
&fld = "FIELD." | &cf;
&ret = SetDBFieldCharDefn(@(&fld), &len, &frmt);
If (&ret = 0) Then
   MessageBox(0, "SetDBFieldCharDefn", 0, 0, "Succeeded");
Else
   MessageBox(0, "SetDBFieldCharDefn", 0, 0, "Failed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetDBFieldLabel , page 411.

SetDBFieldFormat

Syntax

SetDBFieldFormat(**Field**.*FieldName*, *FormatFamily* [, *FamilyName*, *DisplayName*])

Description

Use the **SetDBFieldFormat** function to change the format family for a field.

Use the StoredFormat Field property to determine the existing format family for a field.

If you only want to change the display format of a single field at runtime, and not change the database field, use the DisplayFormat Field property.

Note. This function only works with character fields.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
FieldName	Specify the name of the field that you want to modify. This name must be prefixed by the reserved word **Field**.
FormatFamily	Specify the new format family of the field. The valid values are: • %FormatFamilyType_Upper • %FormatFamilyType_Name • %FormatFamilyType_Phone • %FormatFamilyType_Zip • %FormatFamilyType_SSN • %FormatFamilyType_MixedCase • %FormatFamilyType_NumOnly • %FormatFamilyType_SIN • %FormatFamilyType_PhoneIntl • %FormatFamilyType_ZipIntl • %FomatFamilyType_Custom
FamilyName	Specify a new family name. This parameter is optional, and only valid if *FormatFamily* is specified as custom (%FormatFamilyType_Custom).
DisplayName	Specify a new display name. This parameter is optional, and only valid if *FormatFamily* is specified as custom (%FormatFamilyType_Custom).

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Function completed successfully.
%MDA_Failure	Function didn't complete successfully.

Example

```
&ret = SetDBFieldFormat(Field.OrgId, %FormatFamilyType_Custom, ⇒
"Postal_Code", "Normal");
If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldFormat succeeded");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldFormat failed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetDBFieldFormatLength , page 410, Chapter 12, Field Class , StoredFormat, page 868, Chapter 12, Field Class , DisplayFormat, page 853.

SetDBFieldFormatLength

Syntax

SetDBFieldFormatLength(*FieldName, Length*)

Description

Use the **SetDBFieldFormatLength** function to change the format length for a field. This length controls the maximum number of characters an end-user can type into an edit box for this character field. This can be used to limit the user without having to rebuild or alter the table.

Note. This function only works with character fields.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
FieldName	Specify the name of the field that you want to modify. This name must be prefixed by the reserved word **Field**.
Length	Specify the new format length as a number. Valid values are between 1 and 254.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Function completed successfully.
%MDA_Failure	Function didn't complete successfully.
%MDA_FieldNotFound	The specified field wasn't found in the database.
%MDA_Unsupported	You tried to use this function on a non character field. This function is only supported on character fields.

Example

```
&ret =  SetDBFieldFormatLength(FIELD.OrgId, 10);
If (&ret = %MDA_success) Then
  MessageBox(0, "MetaData Fn Status", 0, 0, "SetDBFieldFormatLength succeeded");
Else
  MessageBox(0, "MetaData Fn Status", 0, 0, "SetDBFieldFormatLength failed");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetDBFieldFormat , page 408, Chapter 12, Field Class , FormatLength, page 855.

SetDBFieldLabel

Syntax

SetDBFieldLabel(**Field**.*FieldName*, *LabelID*, *Long*, *Short*, *Default* [, *LanguageID*])

Description

Use the **SetDBFieldLabel** function to either modify an existing label, or add a new label to a field definition.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
FieldName	Specify the name of the field that you want to modify. This name must be prefixed by the reserved word **Field**.
LabelID	Specify the label ID of the field label that you want to modify as a string. If the specified label ID isn't found, a new label, with the specified label ID, is created for the field.
Long	Specify the new long label for the field as a string.
Short	Specify the new short label for the field as a string.
Default	Specify whether the new label is the default label for the field. This parameter takes a Boolean value: True, set the label as the default, False, don't.
LanguageID	Specify the three character language code to use with this field. This parameter is optional. If you don't specify a language code, the language of the current user is used.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Function completed successfully.
%MDA_Failure	Function didn't complete successfully.
%MDA_FieldNotFound	The specified field wasn't found.

Example

```
&ret = SetDBFieldLabel(Field.OrgId, "ORGID", "Organization ID", "OrgId", TRUE);
If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldLabel succeeded");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldLabel failed");
End-If;
```

See Also

Chapter 12, Field Class , Label, page 861.

SetDBFieldLength

Syntax

SetDBFieldLength(**Field.***FieldName*, *Length*)

Description

Use the **SetDBFieldLength** function to modify an existing character field to have a new length.

Note. Because using this function changes records that are used to build application tables, you must rebuild (alter) the specified project before these changes can be used.

Use the Length Field class property to find the existing length of a field.

Note. This function only works with character fields.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
FieldName	Specify the name of the field that you want to modify. This name must be prefixed by the reserved word **Field**.
Length	Specify the new field length as a number between 1 and 254.

Note. If a default has been specified for this field in any record, and the size of the default is *greater* than the new size, you must modify the record field separately.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Function completed successfully.
%MDA_Failure	Function didn't complete successfully.
%MDA_Unsupported	The specified field isn't a character field. This function is only supported for character fields.
%MDA_FieldNotFound	The specified field wasn't found.
%MDA_FieldFmtLength	The specified length isn't compatible with the current format family, or there are record field defaults greater than the specified size.

Note. If a default has been specified for this field in any record, and the size of the default is *greater* than the new size, you must modify the record field separately.

Example

```
&ret = SetDBFieldLength(Field.OrgId, 10);
```

```
If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldLength succeeded");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldLength failed");
End-If;
```

You can also use this function with de-referenced parameters, as follows:

```
&ret = SetDBFieldLength(@("FIELD." | FS_CF_UPD_AET.FIELDNAME), FS_CF_UPD_AET.NEW_⇒
CF_LENGTH);
```

See Also

Chapter 12, Field Class , FieldLength, page 855.

SetDBFieldNotUsed

Syntax

SetDBFieldNotUsed(**Field**.*FieldName*, *NotUsed*)

Description

Use the **SetDBFieldNotUsed** function to specify whether a database field is used as a chart field or not.

SetDBFieldNotUsed does the following for a field:

- Specifies whether the field is included in the index when indexes are built for records that contain this field. The column always remains in the table associated with the record.
- Specifies that the field is ignored in Query.
- Specifies that the field is ignored in nVision.

Considerations Using this Function

This function is primarily intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not, in general, supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded. Using this function to modify records in components that have not been loaded, and then loading those components will, while not changing indices, prevent Query and nVision from using the field, and may be used to key display of the field in pages.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
FieldName	Specify the name of the field that you want to modify. This name must be prefixed by the reserved word **Field**.
NotUsed	Specify whether this field is to be used as a chart field. This parameter takes a Boolean value: True, this field is not used, False, this field is used.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Function completed successfully.
%MDA_Failure	Function didn't complete successfully.
%MDA_FieldNotFound	The specified field wasn't found.

Example

```
&ret = SetDBFieldNotUsed(Field.OrgId, True);
If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldNotUsed succeeded");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetDBFieldNotUsed failed");
End-If;
```

See Also

PeopleSoft Application Designer, Appendix F: Performing Bulk Operations.

SetDefault

Syntax

SetDefault([*recordname*.]*fieldname*)

Description

SetDefault sets a field to a null value, so that the next time default processing occurs, it is set to its default value: either a default specified in its record field definition or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

Note. This function remains for backward compatibility only. Use the **SetDefault** Field class property instead.

Blank numbers correspond to zero on the database. Blank characters correspond to a space on the database. Blank dates and long characters correspond to NULL on the database. **SetDefault** gives each field data type its proper value.

See Chapter 12, Field Class , SetDefault, page 851.

Where to Use SetDefault

If a PeopleCode program or function executes the SetDefault built-in on a field that does *not* exist in the component buffer, the remainder of the program or function is skipped. In the case of a function, execution of the calling program continues with the next statement after the call to the function. However, if the program containing the SetDefault call is at the "top level", meaning that it was called directly from the component processor or application engine runtime, it exits.

Therefore, if you want to control the behavior of SetDefault, you should encapsulate any calls to this built-in function inside your own functions. This enables your overall programs to continue, whether or not the SetDefault succeeds.

Parameters

Parameter	Description
[recordname.]fieldname	Specify a field designating the fields that you want to set to its default value.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

This example resets the PROVIDER to its null value. This field is reset to its default value when default processing is next performed:

```
If COVERAGE_ELECT = "W" Then
   SetDefault(PROVIDER);
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetDefaultAll , page 416, Chapter 1, PeopleCode Built-in Functions, SetDefaultNext , page 417, Chapter 1, PeopleCode Built-in Functions, SetDefaultPrior , page 418, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Default Processing, *PeopleCode Developer's Guide*, Data Buffer Access.

SetDefaultAll

Syntax

SetDefaultAll([*recordname*.]*fieldname*)

Description

SetDefaultAll sets all occurrences of the specified *recordname.fieldname* within a scroll to a blank value, so that the next time default processing is run these fields are set to their default value, as specified by the record definition, or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

Note. This function remains for backward compatibility only. Use the **SetDefault** Rowset method instead.

See Also

Chapter 27, Rowset Class , SetDefault, page 1499, Chapter 12, Field Class , SearchDefault, page 865, *PeopleCode Developer's Guide*, Data Buffer Access.

Example

The following example sets the fields TO_CUR and CUR_EXCHNG_RT to their default values on every row of the scroll area where the PeopleCode is run:

```
SetDefaultAll(TO_CUR);
```

```
SetDefaultAll(CUR_EXCHNG_RT);
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetDefault , page 415, Chapter 1, PeopleCode Built-in Functions, SetDefaultNext , page 417, Chapter 1, PeopleCode Built-in Functions, SetDefaultNextRel , page 417, Chapter 1, PeopleCode Built-in Functions, SetDefaultPrior , page 418, Chapter 1, PeopleCode Built-in Functions, SetDefaultPriorRel , page 418.

SetDefaultNext

Syntax

SetDefaultNext([*recordname.*]*fieldname*)

Description

SetDefaultNext locates the next occurrence of the *recordname.fieldname* with the next effective date (and effective-sequence number if specified) and sets the field to a blank value, so that the next time default processing is run this field will be set to its default value, as specified by the record definition, or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

SetDefaultNext is typically used to reset values within a scroll which are calculated within default PeopleCode based on a next value.

This function is valid only for effective-dated records. If a next record does not exist, then the statement is skipped.

See Also

Chapter 1, PeopleCode Built-in Functions, SetDefaultAll , page 416, Chapter 1, PeopleCode Built-in Functions, SetDefaultNextRel , page 417, Chapter 1, PeopleCode Built-in Functions, SetDefaultPrior , page 418, Chapter 1, PeopleCode Built-in Functions, SetDefaultPriorRel , page 418.

SetDefaultNextRel

Syntax

SetDefaultNextRel(*search_field, default_field*)

Description

SetDefaultNextRel locates the next occurrence of the *search_field* with the next effective date (and effective-sequence number if the record contains an effective-sequence number) and then sets the value of the specified *default_field* corresponding to the *search_field* to a blank value, so that the next time default processing is run this field will be set to its default value, as specified by the record definition, or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

This function is valid only for effective-dated records. If a next record does not exist, then the statement is skipped.

See Also

Chapter 1, PeopleCode Built-in Functions, SetDefault , page 415, Chapter 1, PeopleCode Built-in Functions, SetDefaultAll , page 416, Chapter 1, PeopleCode Built-in Functions, SetDefaultPrior , page 418, Chapter 1, PeopleCode Built-in Functions, SetDefaultPriorRel , page 418.

SetDefaultPrior

Syntax

SetDefaultPrior([*recordname.*]*fieldname*)

Description

SetDefaultPrior locates the prior occurrence of the *recordname.fieldname* with the prior effective date (and effective-sequence number if specified) and sets the field to a blank value, so that the next time default processing is run this field will be set to its default value, as specified by the record definition, or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

SetDefaultPrior is typically used to reset values within a scroll which are calculated within FieldDefault PeopleCode based on a next value.

This function is valid only for effective-dated records. If a prior record does not exist, then the statement is skipped.

See Also

Chapter 1, PeopleCode Built-in Functions, SetDefault , page 415, Chapter 1, PeopleCode Built-in Functions, SetDefaultAll , page 416, Chapter 1, PeopleCode Built-in Functions, SetDefaultNext , page 417, Chapter 1, PeopleCode Built-in Functions, SetDefaultNextRel , page 417, Chapter 1, PeopleCode Built-in Functions, SetDefaultPriorRel , page 418.

SetDefaultPriorRel

Syntax

SetDefaultPriorRel(*search_field, default_field*)

Description

SetDefaultPriorRel locates the prior occurrence of the *search_field* with the prior effective date (and effective sequence-number if the record contains an effective-equence number) and then sets the specified *default_field* to a blank value, so that the next time default processing is run this field will be set to its default value, as specified by the record definition, or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

This function is valid only for effective-dated records. If a next record does not exist, then the statement is skipped.

See Also

Chapter 1, PeopleCode Built-in Functions, SetDefault , page 415, Chapter 1, PeopleCode Built-in Functions, SetDefaultAll , page 416, Chapter 1, PeopleCode Built-in Functions, SetDefaultNext , page 417, Chapter 1, PeopleCode Built-in Functions, SetDefaultNextRel , page 417, Chapter 1, PeopleCode Built-in Functions, SetDefaultPrior , page 418.

SetDisplayFormat

Syntax

```
SetDisplayFormat(scrollpath, target_row,
          [recordname.]fieldname, display_format_name)
```

where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_
row, ]] RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

Use **SetDisplayFormat** to change the display format of Custom Defined Fields at runtime. For instance, you may want to update a custom numeric display to reveal more decimal points.

Note. This function remains for backward compatibility only. Use the **DisplayFormat** Field property instead.

See Also

Chapter 12, Field Class , DisplayFormat, page 853, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number of the target row.
[*recordname* **.**]*fieldname*	The name of the field to change. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the call to **SetDisplayFormat** is in a record definition other than *recordname*.
display_format_name	The name of the custom display format specified in the field definition.

Returns

Returns a Boolean value indicating whether the function executed successfully. The return value is not optional.

See Also

Chapter 1, PeopleCode Built-in Functions, GetStoredFormat , page 273.

SetLabel

Syntax

```
SetLabel(scrollpath, target_row, [recordname.]fieldname, new_label_text)
```

Where *scrollpath* is:

```
[RECORD.level1_recname, level1_row, [RECORD.level2_recname, level2_
row, ]] RECORD.target_recname
```

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

SetLabel enables you to change the label text of a page field or grid column heading.

Note. This function remains for backward compatibility only. Use the **Label** Field property instead.

You can't use this function to set labels longer than 100 characters. If you try to set a label of more than 100 characters, the label is truncated to 100 characters.

See Also

Chapter 12, Field Class , Label, page 861, Chapter 12, Field Class , GetLongLabel, page 847, Chapter 12, Field Class , GetShortLabel, page 849, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number of the target row.
[*recordname* **.**]*fieldname*	The name of the field with the associated label text. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the function call is in a record definition other than *recordname*.
new_label_text	A String value specifying the new value for the field or grid column label.

Returns

Optionally returns a Boolean value indicating whether the function completed successfully.

Example

```
If training_loc = "HAW" then
     SetLabel(voucher_tbl.training_loc, "Hawaii Training Center");
End-if;
```

SetLanguage

Syntax

```
SetLanguage(language_code)
```

Description

SetLanguage sets the end-user's current language preference to the specified *language_code*. *language_code* must be a valid translate value for the field LANGUAGE_CD. **SetLanguage** returns True if it is successful, and it returns False if it fails or an invalid value was passed. The language preference is temporary, remaining in effect only until the end-user logs off, or until another call is made to **SetLanguage**.

Note. SetLanguage does *not* work in Signon PeopleCode.

Considerations Using SetLanguage With %Language

The value of %Language depends on the type of application:

- For online applications, %Language is the language code that the current component is using.
- For non-online applications (such as batch or in a message subscription), %Language is the language code of the end-user.

Note. Language can also be changed using the Language menu (in the Windows client).

SetLanguage changes the default language *for the end-user*.

%Language reflects the value using SetLanguage after the function is executed.

Parameters

Parameter	Description
language_code	A valid language code, stored in the Translate table for the LANGUAGE_CD field.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully. Returns False if an invalid language code is passed.

Example

The following example switches the language code and displays a message informing the end-user of the change:

```
If %Page = PAGE.INTERNATL_PREF Then
   If SetLanguage(LANGUAGE_CD) Then
      WinMessage(MsgGet(102, 5, "Language preference changed to ", LANGUAGE_CD));
   Else
      WinMessage(MsgGet(102, 6, "Error in setting language.  Language is currently⇒
 %1", %Language));
   End-if;
End-if;
```

See Also

Chapter 3, System Variables , %Language , page 569.

SetNextPanel

Syntax

SetNextPanel(*panelname*)

Description

SetNextPanel specifies the panel name to which the user will be transferred when selecting the NextPanel (F6) function or specifying it with the PeopleCode TransferPage function.

Note. The **SetNextPanel** function is supported for compatibility with previous releases of PeopleTools. New applications should use **SetNextPage** instead.

See Also

Chapter 1, PeopleCode Built-in Functions, SetNextPage , page 422.

SetNextPage

Syntax

SetNextPage(*pagename*)

Description

SetNextPage specifies the page name to which the user is transferred when selecting the NextPage (ALT+6 and ENTER) function or specifying it with the PeopleCode **TransferPage** function.

SetNextPage validates that *pagename* is listed on current menu. This selection is cleared when the user transfers to a new page.

Parameters

Parameter	Description
pagename	A String equal to the name of the page as specified in the page definition.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

See Chapter 1, PeopleCode Built-in Functions, AddKeyListItem , page 36.

```
ClearKeyListItem( );
AddKeyListItem(OPRID, OPRID);
AddKeyListItem(REQUEST_ID, REQUEST_ID);
SetNextPage("PAGE_2");
DoSave( );
TransferPage( );
```

The following example sets up and transfers the user to page JOB_DATA.

```
If SetNextPage(PAGE.JOB_DATA) Then
```

```
        TransferPage( );
    End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475.

SetPageFieldPageFieldName

Syntax

SetPageFieldPageFieldName(**Page.***PageName*, **Record.***RecordName*, **Field.***FieldName, PageFieldName*)

Description

Use the **SetPageFieldPageFieldName** function to add or change a page field name for a field. The page field name is set on the General tab of the page field properties. Changing a name to itself is not supported.

The first field on the page with the specified record name and field name is the field that's changed.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
PageName	Specify the page containing the field you want to change. This name must be prefixed by the reserved word **Page**.
RecordName	Specify the record containing the field you want to change. This name must be prefixed by the reserved word **Record**.
FieldName	Specify the name of the field that you want to modify. This name must be prefixed by the reserved word **Field**.
PageFieldName	Specify the page field name that you want associated with the page field as a string.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Function completed successfully.
%MDA_Failure	Function didn't complete successfully.
%MDA_PageNotFound	The specified page wasn't found.

Value	Description
%MDA_PageFieldNotFound	The specified field wasn't found on the specified page.
%MDA_Duplicate	A second field by the same name was found on the page. Only the first page field name was changed.

Example

The following example adds a page field name to a page field.

```
&ret = SetPageFieldPageFieldName(Page.ABSENCE_HIST, Record.ABSENCE_HIST,⇒
 Field.EMPLID, "EMPLID")
```

The following example adds a page field name to a page field using dereferenced parameters.

```
&Pnl = "Page." | "ABSENCE_HIST";
&Rec = "Record." | "ABSENCE_HIST";
&Field = "Field." | "EMPLID";
&Name = "EMPLID"
&ret = SetPageFieldPageFieldName(@(&Pnl), @(&Rec), @(&Field), &Name);
```

See Also

PeopleSoft Application Designer, Creating Field Definitions.

SetPasswordExpired

Syntax

SetPasswordExpired(*NewValue*)

Description

The **SetPasswordExpired** function sets the password expired status for the current user. When the user's password expired flag is set to True, they can only access the page that allows them to change their password. The function returns the old value, that is, the value that represented the status of the flag before it was set to *NewValue*.

Parameters

Parameter	Description
NewValue	Specify a new value for the user's password expired flag. This parameter takes a Boolean value

Returns

A Boolean value: True if you've set the password expire flag to False, False if you've set the password expire flag to True.

Example

```
If %PasswordExpired Then
   &NewValue = SetPasswordExpired(True);
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SwitchUser , page 454, Chapter 3, System Variables , %PasswordExpired , page 574.
PeopleTools PeopleBook: Peoplesoft Security

SetPostReport

Syntax

```
SetPostReport()
```

Description

The **SetPostReport** function creates a PostReport object. After you've created this object, you can assign values to its properties, then use the Put method to initiate the posting of the files to the Report Repository.

Parameters

None.

Returns

A reference to a PostReport object.

See Also

Chapter 22, PostReport Class , page 1261

SetRecFieldEditTable

Syntax

```
SetRecFieldEditTable(Record.RecordName, Field.FieldName, EditTable [, TableEditType])
```

Description

Use the **SetRecFieldEditTable** function to set the edit table value for a record field. This overwrites the value for the edit table for the record field. Use the SetEditTable Record method to just set the edit table value at runtime.

If you specify a null value for *EditTable*, and no value is specified for *TableEditType*, the table edit flag is turned off, that is, no prompt table is set for the record field.

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
RecordName	Specify the record containing the field you want to change. This name must be prefixed by the reserved word **Record**.
FieldName	Specify the name of the field that you want to modify. This name must be prefixed by the reserved word **Field**.
EditTable	Specify the name of the edit table record. This name must be prefixed by the reserved word **Record**. If you don't want to specify a record name, specify **Record.""**.
TableEditType	Specify the type of edit table record to be associated with the record field. If you specify a value for *EditTable* (and not a null value) this parameter is required. You can specify either a constant or numeric value for this parameter. Valid values are:

Constant Value	Numeric Value	Used for which types of fields
%EditTableType_NoEdit	0	char, num, date, time, datetime
%EditTableType_Prompt	1	char, num, date, time datetime
%EditTableType_YesNo	2	char
%EditTableType_Translate	3	char fields with length 4 or less only
%EditTableType_OneZero	4	number fields only

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Function completed successfully.
%MDA_Failure	Function didn't complete successfully.
%MDA_RecordNotFound	The specified record wasn't found.
%MDA_RecFieldNotFound	The specified field wasn't found on the record.

Example

```
&ret = SetRecFieldEditTable(RECORD.AbsHist, Field.OrgId, RECORD.EmplId_Tbl, ⇒
%EditTableType_Prompt);
If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetRecFieldEditTable succeeded");
Else
```

```
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetRecFieldEditTable failed");
End-If;
```

SetRecFieldKey

Syntax

SetRecFieldKey(**Record.***RecordName*, **Field.***FieldName*, *Key*)

Description

Use the **SetRecFieldKey** function to specify whether a field on a record is a key field or not.

Use the IsKey Field class property to determine whether or not the field is already a key.

Note. Because performing this operation changes records, you must subsequently rebuild the project (alter tables).

Considerations Using this Function

This function is intended for use during configuration time *only*, before active runtime usage is initiated. Using this function during active runtime is not supported. Changes to data definitions are *not* recognized on currently loaded component. In general, changes aren't recognized until the component is reloaded.

Warning! These operations take place in a separate transaction from the page's save status: the initiation of any of these operations immediately changes the definitions, even if the page is subsequently cancelled.

Parameters

Parameter	Description
RecordName	Specify the record containing the field you want to change. This name must be prefixed by the reserved word **Record**.
FieldName	Specify the name of the field that you want to modify. This name must be prefixed by the reserved word **Field**.
Key	Specify whether the field is a key or not. This parameter takes a Boolean value: True, the field is a key field, False, it isn't.

Returns

A constant value. The values are:

Value	Description
%MDA_Success	Function completed successfully.
%MDA_Failure	Function didn't complete successfully.
%MDA_RecordNotFound	The specified record wasn't found.
%MDA_RecFieldNotFound	The specified field wasn't found on the record.

Example

```
&ret = SetRecFieldKey(RECORD.AbsHist, Field.OrgId, True);
If (&ret = %MDA_Success) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetRecFieldKey succeeded");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "SetRecFieldKey failed");
End-If;
```

See Also

PeopleSoft Application Designer, Creating Field Definitions.

SetReEdit

Syntax

SetReEdit(*reedit_on*)

Description

SetReEdit switches re-edit mode on and off. When re-edit mode is *on*, definitional edits (such as translate table and prompt table edits), as well as FieldEdit PeopleCode, are run on each editable field in the component when the component is saved. If an error is found, the component data is not saved. **SetReEdit** can be called at any time during the life of the component before the SaveEdit event fires, and would typically be called in RowInit when other page settings are being initialized. When a component is started, re-edit mode is *off* by default.

SetReEdit is used primarily in financial applications, where transactions are sometimes brought into the database by non-online processes. When re-edit mode is *on*, the values read in during these transactions can be validated by simply bringing them up in the page and saving. Any errors are then reported, as if the end-user had entered all of the data online.

Parameters

Parameter	Description
reedit_on	A Boolean value specifying whether to switch re-edit mode on or off. True turns re-edit mode on, False turns re-edit mode off.

Example

This example is used in RowInit PeopleCode to initialize component settings. After re-edit mode is on, field-level edits are re-applied when the component is saved.

```
SetReEdit(True);
```

SetSearchDefault

Syntax

SetSearchDefault([*recordname.*]*fieldname*)

Description

SetSearchDefault enables system defaults (default values set in record field definitions) for the specified field on search dialog boxes. It does not cause the FieldDefault event to fire.

Note. This function remains for backward compatibility only. Use the **SearchDefault** Field property instead.

The system default occurs only once, when the search dialog box first starts, immediately after SearchInit PeopleCode. If the end-user subsequently blanks out a field, the field is not reset to the default value. The related function **ClearSearchDefault** disables default processing for the specified field. **SetSearchDefault** is effective only when used in SearchInit PeopleCode programs.

See Also

Chapter 12, Field Class , SearchDefault, page 865, *PeopleCode Developer's Guide*, Data Buffer Access.

Parameters

Parameter	Description
[*recordname* **.**]*fieldname*	The name of the target field, a search or alternate search key that is about to appear in the search dialog box. The *recordname* prefix is required if the call to **SetSearchDefault** is in a record definition other than *recordname*.

Example

This example, from SearchInit PeopleCode turns on edits and defaults for the SETID field in the search dialog box:

```
SetSearchEdit(SETID);
SetSearchDefault(SETID);
```

See Also

Chapter 1, PeopleCode Built-in Functions, ClearSearchDefault , page 68, Chapter 1, PeopleCode Built-in Functions, ClearSearchEdit , page 69, Chapter 1, PeopleCode Built-in Functions, SetSearchDialogBehavior , page 429, Chapter 1, PeopleCode Built-in Functions, SetSearchEdit , page 430, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Search Processing in Update Modes.

SetSearchDialogBehavior

Syntax

SetSearchDialogBehavior(*force_or_skip*)

Description

SetSearchDialogBehavior can be called in SearchInit PeopleCode to set the behavior of search and add dialog boxes before a page is displayed, overriding the default behavior. There are two dialog behavior settings: **skip if possible** (0) and **force display** (1).

Skip if possible means that the dialog box is skipped if all of the following are true:

- All required keys have been provided (either by system defaults or by PeopleCode).

- If this an Add dialog box, then no "duplicate key error" results from the provided keys; if this error occurs, the processing resets to the default behavior.
- If this is a Search dialog box, then at least one row is returned based on the provided keys.

Force display means that the dialog box displays even if all required keys have been provided.

The default behavior of the search and add dialog boxes is **force display**.

Note. SetSearchDialogBehavior can only be used in SearchInit PeopleCode.

Parameters

Parameter	Description
force_or_skip	A Number equal to one of the following values: • 0: sets the dialog behavior to **skip if possible**. • 1: sets the dialog behavior to **force display**.

Returns

None.

Example

The following function call, which must occur in SearchInit PeopleCode, sets the dialog behavior to **skip if possible**.

```
SetSearchDialogBehavior(0);
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetSearchDefault , page 428, Chapter 1, PeopleCode Built-in Functions, SetSearchEdit , page 430, Chapter 1, PeopleCode Built-in Functions, ClearSearchEdit , page 69, Chapter 1, PeopleCode Built-in Functions, ClearSearchDefault , page 68, Chapter 1, PeopleCode Built-in Functions, IsUserInRole , page 312, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Search Processing in Update Modes.

SetSearchEdit

Syntax

SetSearchEdit([*recordname.*]*fieldname*)

Description

SetSearchEdit enables system edits (edits specified in the record field definition) for the specified [*recordname.*]*fieldname*, for the life of the search dialog box, or until **ClearSearchEdit** is called with that same field.

Note. This function remains for backward compatibility only. Use the **SearchEdit** Field property instead.

See Chapter 12, Field Class , SearchEdit, page 865.

Using SetSearchEdit

In the Add mode search dialog, the following edits are performed when the end-user clicks the **Add** button. In any other mode, the following edits are performed when the end-user clicks the **Search** button:

- Formatting
- Required Field
- Yes/No Table
- Translate Table
- Prompt Table

SetSearchEdit does not cause the FieldEdit, FieldChange, or SaveEdit PeopleCode events to fire during the search dialog.

You might use **SetSearchEdit** to control access to the system. For example, you can apply this function to the SETID field of a dialog box and require the end-user to enter a valid SETID.

If you use this function in the SearchInit event, the search page options are limited to the "=" and "IN" operators.

Parameters

Parameter	Description
fieldname	The name of the search dialog field on which to enable field edits.

Returns

Returns a Boolean value indicating whether the function executed successfully.

Example

This example turns on edits and system defaults for the SETID field in the search dialog box:

```
SetSearchEdit(SETID);
SetSearchDefault(SETID);
```

See Also

Chapter 1, PeopleCode Built-in Functions, ClearSearchEdit , page 69, Chapter 1, PeopleCode Built-in Functions, ClearSearchDefault , page 68, Chapter 1, PeopleCode Built-in Functions, SetSearchDefault , page 428, Chapter 1, PeopleCode Built-in Functions, SetSearchDialogBehavior , page 429, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Search Processing in Update Modes, *PeopleCode Developer's Guide*, Data Buffer Access.

SetTempTableInstance

Syntax

SetTempTableInstance(*instance_number*)

Description

The **SetTempTableInstance** function sets the default temp table instance to the specified number for the processing of temporary tables. This default is used by all **%Table** meta-SQL references to temporary tables, and by all SQL operations. Generally, you use this function only when you're trying to use any of the ScrollSelect functions, the Rowset class Select or SelectAll methods, the Record class SQL methods (SelectByKey, Insert, and so on.), or any of the meta-SQL statements that use %Table (%InsertSelect, %SelectAll, %Delete, and so on.) Generally, %Table should be used to override the default.

If you use this built-in within an Application Engine program, and the program uses a process-level instance on the request, the old instance value *must be* saved, then restored after you're finished using the new instance.

If you pass a zero for *instance_number*, the **Fill** method uses the physical table instance with no table append, for example, if the temporary table record is FI_INSTR_T, the physical table used is PS_FI_INSTR_T.

Parameters

Parameter	Description
instance_number	Specify the instance number for the temporary tables.

Returns

Existing (or previous) instance number.

Example

To avoid interfering with other uses of temporary tables, you should only set the temporary table instance for your process, then set it back to the default. For example:

```
/* Set temp table instance */
&PrevInstNum = SetTempTableInstance(&NewInstNum);
/* use the temporary table */
.  .  .
/* Restore the temp table instance */
SetTempTableInstance(&PrevInstNum);
```

See Also

PeopleSoft Application Engine, Using Meta-SQL and PeopleCode, %Table.

SetTracePC

Syntax

SetTracePC(*n*)

Description

SetTracePC enables you to control Trace PeopleCode settings programmatically. This is useful if you want to isolate and debug a single program or part of a program.

Note. If you're using an API with the Session class, use the Trace Setting class properties instead of this function.

You can set options prior to starting a PeopleTools session using the Trace tab of PeopleSoft Configuration Manager.

Trace PeopleCode creates and writes data to a trace file that it shares with Trace SQL; Trace SQL and Trace PeopleCode information are both output to the file in the order of execution. The trace file uses a file name and location specified in the Trace page of PeopleSoft Configuration Manager. If no trace file is specified in PeopleSoft Configuration Manager, the file is set by default to DBG1.TMP in your Windows Temp directory. If you specify only a file name, and no directory is specified, the file is written to the directory you're running Tools from. This file is cleared each time you log on and can be opened in a text editor while you are in a PeopleTools session, so if you want to save it, you must print it or copy it from your text editor.

Note. PeopleSoft recommends using a value of %TracePC_EachStmt (2048) instead of %TracePC_Functions (1) and %TracePC_List (2).

See Also

Chapter 29, Session Class , Trace Setting Class Properties , page 1580, *PeopleSoft Administration Tools*, PeopleSoft Configuration Manager, Understanding PeopleSoft Configuration Manager.

Parameters

Parameter	Description
options	Either a Number or a constant value that sets trace options. Calculate *options* by either totaling the numbers associated with any of the following options or by adding constants together:

Numeric Value	Constant Value	Description
0	%TracePC_None	Set trace off.
1	%TracePC_Functions	Provide a trace of the program as it is executed. This implies options 64, 128, and 256 described in the following rows.
2	%TracePC_List	Provide a listing of the entire program.
4	%TracePC_Assigns	Show the results of all assignments made to variables.
8	%TracePC_Fetches	Show the values fetched for all variables.
16	%TracePC_Stack	Show the contents of the internal machine stack. This option is normally used for debugging the PeopleCode language and not PeopleCode programs.
64	%TracePC_Starts	Provide a trace showing when each program starts.
128	%TracePC_ExtFuncs	Provide a trace showing the calls made to each external PeopleCode routine.
256	%TracePC_IntFuncs	Provide a trace showing the calls made to each internal PeopleCode routine.
512	%TracePC_ParamsIn	Show the values of the parameters to a function.
1024	%TracePC_ParamsOut	Show the values of the parameters as they exist at the return from a function.

Numeric Value	Constant Value	Description
2048	%TracePC_EachStmt	Show each statement as it's executed (and don't show statements on branches not taken.)

Returns

None.

Example

The following example is part of a SavePreChange PeopleCode program that sets PeopleCode trace based on page field settings:

```
DEBUG_CODE = 0;
If DEBUG_TRACE_ALL = "Y" Then
   DEBUG_CODE = DEBUG_CODE + 1
End-if;
If DEBUG_LIST = "Y" Then
   DEBUG_CODE = DEBUG_CODE + 2
End-if;
If DEBUG_SHOW_ASSIGN = "Y" Then
   DEBUG_CODE = DEBUG_CODE + 4
End-if;
If DEBUG_SHOW_FETCH = "Y" Then
   DEBUG_CODE = DEBUG_CODE + 8
End-if;
If DEBUG_SHOW_STACK = "Y" Then
   DEBUG_CODE = DEBUG_CODE + 16
End-if;
If DEBUG_TRACE_START = "Y" Then
   DEBUG_CODE = DEBUG_CODE + 64
End-if;
If DEBUG_TRACE_EXT = "Y" Then
   DEBUG_CODE = DEBUG_CODE + 128
End-if;
If DEBUG_TRACE_INT = "Y" Then
   DEBUG_CODE = DEBUG_CODE + 256
End-if;
If DEBUG_SHOW_PARMS = "Y" Then
   DEBUG_CODE = DEBUG_CODE + 512
End-if;
If DEBUG_SHOW_PARMSRT = "Y" Then
   DEBUG_CODE = DEBUG_CODE + 1024
End-if;
SetTracePC(DEBUG_CODE);
```

The following example sets Trace PC to show a listing of all the calls made to external routines as well as calls made to internal routines:

```
SetTracePC(384);
```

The following is identical to the previous example:

```
SetTracePC(%TracePC_ExtFuncs + %TracePC_IntFuncs);
```

If you need a thorough trace, you can use a value of 3596. That combines the following:

Value	Description
2048	Show each statement as it's executed
1024	Show the values of the parameters as they return
512	Show the values of the parameters to a function
8	Show the values fetched for all variables
4	Show the results of all assignments

See Also

Chapter 1, PeopleCode Built-in Functions, SetTraceSQL , page 435, *PeopleSoft Server Tools Administration*, Database Level Auditing.

SetTraceSQL

Syntax

SetTraceSQL(*options*)

Description

SetTraceSQL provides programmatic control over the Trace SQL utility, enabling you to control TraceSQL options during the course of program execution.

Note. If you're using an API with the Session class, use the Trace Setting class properties instead of this function.

When you interact with PeopleTools, SQL statements transparently perform actions such as page construction. The Trace SQL utility creates and updates a file showing the SQL statements generated by PeopleTools.

You can set options prior to starting a PeopleTools session using the Trace tab of PeopleSoft Configuration Manager.

Trace SQL creates and writes data to a trace file that it shares with Trace PeopleCode; Trace SQL and Trace PeopleCode information are both output to the file in the order of execution. The trace file uses a file name and location specified in the Trace page of PeopleSoft Configuration Manager. If no trace file is specified in PeopleSoft Configuration Manager, the file is set by default to DBG1.TMP in your Temp directory. If you specify only a file name, and no directory is specified, the file is written to the directory you're running Tools from. This file is cleared each time you log on and can be opened in a text editor while you are in a PeopleTools session, so if you want to save it, you must print it or copy it from your text editor.

See Also

Chapter 29, Session Class , Trace Setting Class Properties , page 1580, *PeopleSoft Administration Tools*, PeopleSoft Configuration Manager, Understanding PeopleSoft Configuration Manager.

Parameters

Parameter	Description
options	Either a Number value or a constant that sets trace options. Calculate *options* by either totaling the numbers associated with any of the following options, or adding constants together:

Numeric Value	Constant Value	Description
0	%TraceSQL_None	No output
1	%TraceSQL_Statements	SQL statements
2	%TraceSQL_Variables	SQL statement variables (binds)
4	%TraceSQL_Connect	SQL connect, disconnect, commit and rollback
8	%TraceSQL_Fetch	Row Fetch (indicates that it occurred and the return code - not the data fetched.)
16	%TraceSQL_MostOthers	All other API calls except Set Select Buffers
32	%TraceSQL_SSB	Set Select Buffers(identifies the attributes of the columns to be selected)
64	%TraceSQL_DBSpecific	Database API-specific calls
128	%TraceSQL_Cobol	COBOL Statement Timings
256	%TraceSQL_SybBind	Sybase Bind Information
512	%TraceSQL_SybFetch	Sybase Fetch Information
1024	%TraceSQL_DB2400Server	Manager information for DB2/400 only
4096	%TraceSQL_ManagerInfo	All manager information.
8192	%TraceSQL_AppEngineInfo	Trace Application Engine information.

Note. PeopleSoft recommends setting options to 3 to provide most essential information without performance degradation. Options 8 and 32 greatly increase the volume of the trace and will noticeably degrade performance.

Returns

None.

Example

The following example switches off Trace SQL:

```
SetTraceSQL(0);
```

The following is identical to the previous example:

```
SetTraceSQL(%TraceSQL_None);
```

The following example sets Trace SQL to typical settings that won't degrade performance:

```
SetTraceSQL(3);
```

The following is identical to the previous example:

```
SetTraceSQL(%TraceSQL_Statements + %TraceSQL_Variables);
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetTracePC , page 432, *PeopleSoft Server Tools Administration*, Database Level Auditing.

SetupScheduleDefnItem

Syntax

SetupScheduleDefnItem(*ScheduleName, JobName*)

Description

The **SetupScheduleDefnItem** function creates a ProcessRequest object. After you've created this object, you can assign values to its properties then specific methods created to either schedule or print info for a Scheduled Jobset.

Parameters

Parameter	Description
ScheduleName	Specify the process type as a string. Values depend on the Scheduled Jobset defined for your system.
JobsName	Specify the name of the job name as a string.

Returns

A reference to a ProcessRequest object.

Example

```
Local ProcessRequest &MYRQST;

&MYRQST = SetupScheduleDefnItem("SampleSchedule", &MyJobName);
```

See Also

Chapter 23, ProcessRequest Class , page 1269, Chapter 23, ProcessRequest Class , Schedule, page 1286, .

SetUserOption

Syntax

SetUserOption(*Level, OPTN, Value*)

Description

Use the **SetUserOption** to set the default value for the specified option.

Parameters

Parameter	Description
Level	Specify the option category level as a string.
OPTN	Specify the option as a string.
Value	Specify the value of the option.

Returns

A Boolean value: True, if the function completed successfully, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, GetUserOption , page 280, *PeopleTools Security*, Managing PeopleSoft Personalizations.

Sign

Syntax

Sign(*n*)

Description

Sign determines the sign of a number.

Parameters

Parameter	Description
n	A Number value of which to determine the sign.

Returns

Returns a Number value equal to:

- 1 if *n* is positive
- 0 if *n* is 0
- -1 if *n* is negative

Example

The example sets &NUMSIGN to 1:

```
&NUMSIGN = Sign(25);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Abs , page 27.

Sin

Syntax

Sin(*angle*)

Description

Sin calculates the sine of the given angle (opposite / hypotenuse).

Parameters

Parameter	Description
angle	A value in radians.

Returns

A real number between -1.00 and 1.00.

Example

The following example returns the sine of an angle measuring 1.2 radians:

```
&MY_RESULT = Sin(1.2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Acos , page 29, Chapter 1, PeopleCode Built-in Functions, Asin , page 45, Chapter 1, PeopleCode Built-in Functions, Atan , page 46, Chapter 1, PeopleCode Built-in Functions, Cos , page 91, Chapter 1, PeopleCode Built-in Functions, Cot , page 92, Chapter 1, PeopleCode Built-in Functions, Degrees , page 134, Chapter 1, PeopleCode Built-in Functions, Radians , page 356, Chapter 1, PeopleCode Built-in Functions, Tan , page 456.

SinglePaymentPV

Syntax

SinglePaymentPV(*int_rate*, *n_per*)

Description

This function calculates the future value of a single monetary unit after a specified number of periods at a specified interest rate.

Parameters

Parameter	Description
int_rate	A Number representing the interest rate at which value is accrued per period.
n_per	A Number specifying the number of periods on which to base the calculated value.

Returns

Returns a Number value equal to the value of the unit after *n_per* periods at an interest rate of *int_rate* per period.

Example

The example calculates &PMT as .857338820301783265:

```
&PMT = SinglePaymentPV(8, 2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, UniformSeriesPV , page 489.

SortScroll

Syntax

SortScroll(*level*, *scrollpath*, *sort_fields*)

Where *scrollpath* is:

[**RECORD.***level1_recname*, [**RECORD.***level2_recname*,] **RECORD**.*target_recname*

and where *sort_fields* is a list of field specifiers in the form:

[*recordname*.]*field_1*, *order_1* [, [*recordname*.]*field_2*, *order_2*]. . .

Description

SortScroll programmatically sorts the rows in a scroll area on the active page. The rows can be sorted on one or more fields.

Note. This function remains for backward compatibility only. Use the **Sort** Rowset method instead.

The type of sort done by this function, that is, whether it is a linguistic or binary sort, is determined by the **Sort Order Option** on the PeopleTools Options page.

See Also

Chapter 27, Rowset Class , Sort, page 1500, *PeopleCode Developer's Guide*, Data Buffer Access.

Parameters

Parameter	Description
level	Integer specifying the level of the scroll to sort. It can be 1, 2, or 3.
scrollpath	A construction that specifies a scroll area in the component buffer.
sort_fields	A list of field and order specifiers which act as sort keys. The rows in the scroll area are sorted by the first field in either ascending or descending order, then by the second field in either ascending or descending order, and so on.
[*recordname.*]*field_n*	Specifies a sort key field in *target_recname*. The recordname prefix is required if the call to **SortScroll** is in a record other than *target_recname*.
order_n	A single-character string. "A" specifies ascending order; "D" specifies descending order.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

The first example repopulates a scroll in a page programmatically by first flushing its contents, selecting new contents using **ScrollSelect**, then sorting the rows in ascending order by EXPORT_OBJECT_NAME:

```
Function populate_scrolls;
   ScrollFlush(RECORD.EXPORT_OBJECT);
   ScrollSelect(1, RECORD.EXPORT_OBJECT, RECORD.EXPORT_OBJECT,
               "where export_type = :EXPORT_TYPE_VW.EXPORT_TYPE");
   SortScroll(1, RECORD.EXPORT_OBJECT, EXPORT_OBJECT.EXPORT_OBJECT_NAME, "A");
End-function;
```

The second example sorts the rows on scroll level one by primary and secondary key fields:

```
SortScroll(1,RECORD.EN_BOM_COMPS,EN_BOM_COMPS.SETID,"A",
   EN_BOM_CMOPS.INV_ITEM_ID,"A");
```

See Also

Chapter 1, PeopleCode Built-in Functions, HideScroll , page 289, Chapter 1, PeopleCode Built-in Functions, RowScrollSelect , page 384, Chapter 1, PeopleCode Built-in Functions, RowScrollSelectNew , page 386, Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389, Chapter 1, PeopleCode Built-in Functions, ScrollSelectNew , page 392, Chapter 1, PeopleCode Built-in Functions, UnhideScroll , page 488, *PeopleSoft Server Tools Administration*, PeopleTools Utilities, PeopleTools Options.

Split

Syntax

Split(*string, separator*)

Description

The **Split** function *converts* a string into an array of strings by looking for the string *separator* in the given string.

Note. Split does not split an array.

If *separator* is omitted, a blank is used. If *separator* is a null string (""), the string is split into single characters.

Parameters

Parameter	Description
string	The string to be converted into an array
separator	The character used for dividing the string.

Returns

Returns a reference to the array.

Example

The following code produces in &AS the array ("This", "is", "a", "simple", "example.").

```
&STR = "This is a simple example.";
```

```
&AS = Split(&STR);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateArray , page 92, Chapter 1, PeopleCode Built-in Functions, CreateArrayRept , page 94, Chapter 7, Array Class , page 633.

SQLExec

Syntax

SQLExec({*sqlcmd* | **SQL.***sqlname*}, *bindexprs*, *outputvars*)

where *bindexprs* is a list of expressions, one for each *:n* reference within *sqlcmd* or the text found in the SQL defintion *sqlname*, in the form:
inexpr_1 [, *inexpr_2*]. . .

and where *outputvars* is a list of variables, record fields, or record object references, one for each column selected by the SQL command, in the form:
out_1 [, *out_2*]. . .

Description

The **SQLExec** function executes a SQL command from within a PeopleCode program by passing a SQL command string. The SQL command bypasses the Component Processor and interacts with the database server directly. If you want to delete, insert, or update a single record, use the corresponding PeopleCode record object method.

If you want to delete, insert, or update a series of records, all of the same type, use the CreateSQL or GetSQL functions then the **Execute** SQL object method.

Limitation of SQLExec SELECT Statement

SQLExec can only SELECT a single row of data. If your SQL statement (or your SQL.*sqlname* statement) retrieves more than one row of data, **SQLExec** sends only the first row to its output variables. Any subsequent rows are discarded. This means if you want to fetch only a single row, SQLExec can perform better than the other SQL functions, because only a single row is fetched. If you need to SELECT multiple rows of data, use the CreateSQL or GetSQL functions and the Fetch method. You can also use ScrollSelect or one of the Select methods on a rowset object to read rows into a (usually hidden) work scroll.

Note. The PeopleSoft record name specified in the SQL SELECT statement must be in uppercase.

Limitations of SQLExec UPDATE, DELETE, and INSERT Statements

SQLExec statements that result in a database update (specifically, UPDATE, INSERT, and DELETE) can only be issued in the following events:

- SavePreChange
- WorkFlow
- SavePostChange
- FieldChange

Remember that **SQLExec** UPDATEs, INSERTs, and DELETEs go directly to the database server, not to the Component Processor (although **SQLExec** can *look at* data in the buffer using bind variables included in the SQL string). If a **SQLExec** assumes that the database has been updated based on changes made in the component, that **SQLExec** can be issued only in the SavePostChange event, because before SavePostChange none of the changes made to page data has actually been written back to the database.

Setting Data Fields to Null

SQLExec does *not* set Component Processor data buffer fields to NULL after a row not found fetching error. However, it does set fields that aren't part of the Component Processor data buffers to NULL. Work record fields are also reset to NULL.

Using Meta-SQL in SQLExec

Different DBMS platforms have slightly different formats for dates, times, and datetimes; and PeopleSoft has its own format for these data types as well. Normally the Component Processor performs any necessary conversions when platform-specific data types are read from the database into the buffer or written from the buffer back to the database.

When an SQLExec statement is executed, these automatic conversions don't take place. Instead, you need to use meta-SQL functions inside the SQL command string to perform the conversions. The basic types of meta-SQL functions are:

- General functions that expand at runtime to give you lists of fields, key fields, record fields, and so on. %InsertSelect or %KeyEqual are typical examples.
- In functions that expand at runtime into platform-specific SQL within the WHERE clause of a SELECT or UPDATE statement or in an INSERT statement. %DateIn is a typical example.
- Out functions that expand at runtime into platform-specific SQL in the main clause of SELECT statement. %DateOut is a typical example.

Following is an example of an SQL SELECT using both and "in" and "out" metastring:

```
select emplid, %dateout(effdt) from PS_CAR_ALLOC a where car_id = '" |⇒
 &REGISTRATION_NO | "' and plan_type = '" | &PLAN_TYPE | "' and a.effdt = ⇒
(select max (b.effdt) from PS_CAR_ALLOC b where a.emplid=b.emplid and b.effdt <=⇒
 %currentdatein) and start_dt <= %currentdatein and (end_dt is null or end_dt >=⇒
 %currentdatein)";
```

See Chapter 2, Meta-SQL , page 511.

Bind Variables in SQLExec

Bind variables are references within the *sqlcmd* string to record fields listed in *bindvars*. Within the string, the bind variables are integers preceded by colons:

```
:1, :2,. . .
```

The integers need not in numerical order. Each of these :*n* integers represents a field specifier in the *bindvars* list, so that :1 refers to the first field reference in *bindvars*, :2 refers to the second field reference, and so on.

For example, in the following statement:

```
SQLExec("Select sum(posted_total_amt)
   from PS_LEDGER
```

```
   where deptid between  :1 and :2", DEPTID_FROM, DEPTID_TO, &SUM);
```

:1 is replaced by the value contained in the record field DEPTID_FROM; :2 is replaced by the value contained in the record field DEPTID_TO.

Note the following points:

- Bind variables can be used to refer to long character (longchar) fields. Long character fields are represented in PeopleCode as strings. You should use %TextIn() Meta-SQL to ensure these fields are represented correctly on all database platforms.
- Bind variables can be passed as parameters to meta-SQL functions, for example:

```
SQLExec(". . .%datein(:1). . .", START_DT, &RESULT)
```

- If a bind variable :*n* is a Date field that contains a null value, **SQLExec** replaces all occurrences of ":n" located before the first WHERE clause with "NULL" and all occurrences of "= :n" located after the first WHERE to "IS NULL".

Inline Bind Variables in SQLExec

Inline bind variables are included directly in the SQL string in the form:

```
:recordname.fieldname
```

The following example shows the same **SQLExec** statement with standard bind variables, then with inline bind variables:

```
Rem without Inline Bind Variables;
SQLExec("Select sum(posted_total_amt)
   from PS_LEDGER
   where deptid between  :1 and :2", deptid_from, deptid_to, &sum);

Rem with Inline Bind Variables;
SQLExec("Select sum(posted_total_amt)
   from PS_LEDGER
   where deptid between  :LEDGER.DEPTID_FROM
   and :LEDGER.DEPTID_TO",  &sum);
```

Inline bind variables, like all field and record references enclosed in strings, are considered by PeopleTools as a "black box". If you rename records and fields, PeopleTools does not update record and field names that are enclosed in strings as inline bind variables. For this reason, you should use standard bind variable in preference to inline bind variables wherever standard bind variables are available (as they are in **SQLExec**).

Prior to PeopleTools 8.0, PeopleCode replaced runtime parameter markers in SQL strings with the associated literal values. For databases that offer SQL statement caching, a match was never found in the cache so the SQL had to be re-parsed and re-assigned a query path. However, with PeopleTools 8.0, PeopleCode passes in bind variable parameter markers. For databases with SQL caching, this can offer significant performance improvements.

If you use inline bind variables, they will still be passed as literals to the database. However, if you convert them to bind variables, you may see significant performance improvements.

Output Variables in SQLExec

If you use **SQLExec** to SELECT a row of data, you must place the data into variables or record fields so that it can be processed. You list these variables or fields, separated by commas in the *output* part of the statement following the *bindvars* list. Supply one variable or field for each column in the row of data retrieved by **SQLExec**. They must be listed in the same order in which the columns will be selected.

The number of output variables cannot exceed 64.

Using Arrays for Bind Variables

You can now use a parameter of type "Array of Any" in place of a list of bind values or in place of a list of fetch result variables. This is generally used when you don't know how many values are needed until the code runs.

For example, suppose that you had some PeopleCode that dynamically (that is, at runtime) generated the following SQL statement:

```
&Stmt = "INSERT INTO PS_TESTREC (TESTF1, TESTF2, TESTF3, TESTF4, . . . N) VALUES (:⇒
1, :2, %DateTimeIn(:3), %TextIn(:4), . . .N)";
```

Suppose you have placed the values to be inserted into an Array of Any, say &AAny:

```
&AAny = CreateArrayAny("a", 1, %DateTime, "abcdefg", . . .N);
```

You can execute the insert by:

```
SQLExec(&Stmt, &AAny);
```

Because the Array of Any promotes to absorb any remaining select columns, it must be the last parameter for the SQL object Fetch method or (for results) SQLExec. For binding, it must be the only bind parameter, as it is expected to supply all the bind values needed.

SQLExec Maintenance Issues

SQLExec statements are powerful, but they can be difficult to upgrade and maintain. If you use a SQL string passed in the command, it's considered a "black box" by PeopleCode. If field names or table names change during an upgrade, table and field references within the SQL string are not updated automatically. For these reasons, you should use a SQL definition and the meta-SQL statements provided in PeopleTools 8.0, instead of typing in a SQL string.

Generally, you should use **SQLExec** only when you must interact directly with the database server and none of the **ScrollSelect** functions, or record object methods (which are somewhat easier to maintain) will serve your purpose effectively.

Be Careful How You Use It

SQLExec performs any SQL statement the end-user has database privileges to perform. This normally includes SELECT, INSERT, UPDATE, and DELETE statements against application data tables. However, you can set up end-users with more privileges (and you have to on some platforms). In such cases, the end-user could alter the structure of tables using **SQLExec**, or even drop the database!

Warning! The PeopleSoft application will not stop the end-user from doing anything that the end-user has privileges to do on the database server, so be very careful what you write in an SQLExec statement.

Parameters

Parameter	Description
sqlcmd | SQL.*sqlname*	Specify either a String containing the SQL command to be executed or a reference to an existing SQL definition. This string can include bind variables, inline bind variables, and meta-SQL.
bindexprs	A list of expressions, each of which corresponds to a numeric (:*n*) bind variable reference in the SQL command string. It can also be a reference to a record object or an array of Any containing all the bind values. See Bind Variables in SQLEXEC for more details.
outputvars	A list of PeopleCode variables or record fields to hold the results of a SQL SELECT. There must be one variable for each column specified in the SELECT statement. It can also be a reference to a record object or an Array of Any that contains all the selected values.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

The following example, illustrates a SELECT statement in a SQLExec:

```
SQLExec("SELECT COUNT(*) FROM PS_AE_STMT_TBL WHERE AE_PRODUCT = :1 AND AE_APPL_ID =⇒
 :2 AND AE_ADJUST_STATUS = 'A' ", AE_APPL_TBL.AE_PRODUCT, AE_APPL_TBL.AE_APPL_ID,⇒
 AE_ADJ_AUTO_CNT);
```

Note the use of bind variables, where :1 and :2 correspond to AE_APPL_TBL.AE_PRODUCT and AE_APPL_TBL.AE_APPL_ID. AE_ADJ_AUTO_CNT is an output field to hold the result returned by the SELECT.

The next example is also a straightforward SELECT statement, but one which uses the %datein meta-SQL function, which expands to appropriate platform-specific SQL for the :5 bind variable:

```
SQLExec("SELECT 'X', AE_STMT_SEG FROM PS_AE_STMT_B_TBL where AE_PRODUCT = :1 AND⇒
 AE_APPL_ID = :2 AND AE_SECTION = :3 AND DB_PLATFORM = :4 AND EFFDT = %datein(:5)⇒
 AND AE_STEP = :6 AND AE_STMT_TYPE = :7 AND AE_SEQ_NUM = :8", AE_STMT_TBL.AE_⇒
PRODUCT, AE_STMT_TBL.AE_APPL_ID, AE_STMT_TBL.AE_SECTION, AE_STMT_TBL.DB_PLATFORM,⇒
 AE_STMT_TBL.EFFDT, AE_STMT_TBL.AE_STEP, AE_STMT_TBL.AE_STMT_TYPE, &SEG, &EXIST,⇒
 &STMT_SEG);
```

This last example (in SavePreChange PeopleCode) passes an INSERT INTO statement in the SQL command string. Note the use of a date string this time in the %datein meta-SQL, instead of a bind variable:

```
SQLExec("INSERT INTO PS_AE_SECTION_TBL ( AE_PRODUCT, AE_APPL_ID, AE_SECTION, DB_⇒
PLATFORM, EFFDT, EFF_STATUS, DESCR, AE_STMT_CHUNK_SIZE, AE_AUTO_COMMIT, AE_⇒
SECTION_TYPE ) VALUES ( :1, :2, :3, :4, %DATEIN('1900-01-01'), 'A', ' ', 200,⇒
 'N', 'P' )", AE_APPL_TBL.AE_PRODUCT, AE_APPL_TBL.AE_APPL_ID, AE_SECTION, DB_⇒
PLATFORM);
```

In the following example, a SQLExec statement is used to select into a record object.

```
Local Record &DST;
```

```
&DST = CreateRecord(RECORD.DST_CODE_TBL);
&DST.SETID.Value = GetSetId(FIELD.BUSINESS_UNIT, DRAFT_BU, RECORD.DST_CODE_TYPE,⇒
 "");
&DST.DST_ID.Value = DST_ID_AR;
SQLExec("%SelectByKeyEffDt(:1,:2)", &DST, %Date, &DST);
/* do further processing using record methods and properties */
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109, Chapter 1, PeopleCode Built-in Functions, FetchSQL , page 180, Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271, Chapter 1, PeopleCode Built-in Functions, StoreSQL , page 450, Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389. Chapter 31, SQL Class , page 1613

Sqrt

Syntax

SQRT(*n*)

Description

Sqrt calculates the square root of a number.

Parameters

Parameter	Description
n	A Number of which you want to find the square root.

Returns

Returns a Number equal to the positive square root of *n*. If *n* is a negative number, **Sqrt** displays an error.

Example

The examples return 15, 4, and 8.42615, respectively:

```
&NUM = Sqrt(225);
&NUM = Sqrt(16);
&NUM = Sqrt(71);
```

StartWork

Syntax

StartWork()

Description

Use the **StartWork** function to mark the start of a unit of work.

Once this function is executed, no updates to the database are allowed until a *unit of work* is completed.

A unit of work is completed by an event completing (such as a FieldChange event) in which case all the Updates are saved.

A unit of work can also be completed using the CommitWork built-in function.

If a SQL failure occurs anytime during the unit of work, after the StartWork function has been called and before the unit of work completes, all updates are rolled back, up to when the StartWork function was executed.

This function can be used for nested component interface calls, such that if the lower level component interface fails, any database changes made by the calling component interface can be rolled back.

Parameters

None.

Returns

None.

Example

```
&oCI = &SESSION.GetCompIntfc(CompIntfc.CUSTOMER);

   If &oCI <> Null Then
      .
      .
      .
      For &i = 1 To &rsCustomer.RowCount
         &recCust = &rsCustomer(&Transaction).GetRecord(Record.CUSTOMER);
         StartWork();
         If &oCI.Create() Then
             rem ***** Set CI Properties *****;
      .
         .
         .
             If Not &oCI.Save() Then
                 rem ***** Error Handling *****;
                 .....
             End-If;
         End-If;

         rem ***** CommmitWork ensures that all transactions between     *****;
         rem ***** StartWork and CommitWork get committed to the database *****;

         CommitWork();

         &oCI.Cancel();
      .
      .
      .
      End-For;
   End-If
```

See Also

Chapter 10, Component Interface Classes , page 773.

StopFetching

Syntax

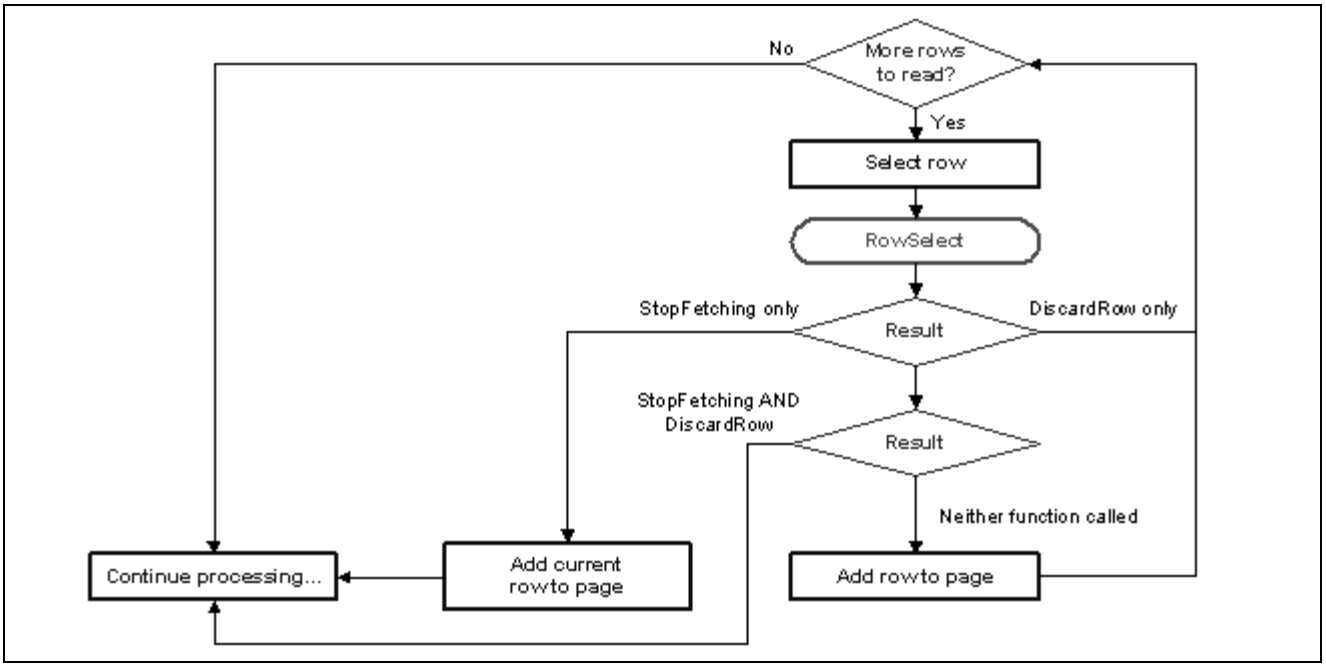
```
StopFetching()
```

Description

StopFetching is called during Row Select processing, during which rows of data that have been selected down from the database can be filtered as they are added to the component. This function is valid only in RowSelect PeopleCode. If **StopFetching** is called without **DiscardRow**, it adds the current row to the component, then stops adding any more rows. If StopFetching is called with **DiscardRow**, the system skips the current row and stops adding rows to the component.

StopFetching has the same functionality as **Error** in the RowSelect event. The anomalous behavior of Error is supported for compatibility with previous releases of PeopleTools.

Note. Row Select processing is used infrequently, because it is more efficient to filter out rows of data using a search view or an effective-dated record before the rows are selected down to the client from the database server.



Flow of Row Select processing

Returns

None.

See Also

Chapter 1, PeopleCode Built-in Functions, DiscardRow , page 146, Chapter 1, PeopleCode Built-in Functions, Error , page 165, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Row Select Processing.

StoreSQL

Syntax

```
StoreSQL(sqlstring, [SQL.]sqlname[, dbtype[, effdt]])
```

Description

The **StoreSQL** function writes the given *sqlstring* value to a SQL definition, storing it under the name *sqlname*, with the database type *dbtype* and the effective date *effdt*. If *sqlname* is a literal name, it must be in the form SQL.*sqlname* or in quotes ("*sqlname*").

To specify a generic statement, that is, one that is overridden by any other matching statement, *dbtype* as **Default** and *effdt* as the null date (or Date(19000101).

You must commit all database changes prior to using this function. This is to avoid locking critical Tools tables and hence freezing all other users. You receive a runtime error message if you try to use this function when there are pending database updates, and your PeopleCode program terminates. You need to commit any database updates prior to using this function. The CommitWork PeopleCode function has been enhanced to allow this.

Parameters

Parameter	Description
sqlstring	Specify the SQL string to be saved as the SQL definition. This parameter takes a string value.
sqlname	Specify the name of the SQL definition to be created. This is either in the form SQL.*sqlname* or a string value giving the *sqlname*.
dbtype	Specify the database type to be associated with the SQL definition. This parameter takes a string value. If *dbtype* isn't specified or is null (""), it is set by default to the current database type (the value returned from the %DbName system variable.) Values for *dbtype* are as follows. These values are not case-sensitive: • APPSERVER • DB2 • DB2UNIX • INFORMIX • MICROSOFT • ORACLE • SYBASE **Note.** Note. Database platforms are subject to change.
effdt	Specify the effective date to be associated with the SQL definition. If *effdt* isn't specified, it is set by default to the current as of date, that is, the value returned from the %AsOfDate system variable.

Returns

None.

Example

The following code stores the select statement as a SQL definition under the name SELECT_BY_EMPLID, for the current database type and effective as of the current as of date:

```
StoreSQL("%Select(:1) where EMPLID = :2", SQL.SELECT_BY_EMPLID);
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109, Chapter 1, PeopleCode Built-in Functions, DeleteSQL , page 141, Chapter 1, PeopleCode Built-in Functions, FetchSQL , page 180, Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442, Chapter 1, PeopleCode Built-in Functions, CommitWork , page 71, Chapter 3, System Variables , %DbName , page 567, Chapter 3, System Variables , %AsOfDate , page 563.
Chapter 31, SQL Class , page 1613

String

Syntax

String(*value*)

Description

String converts any non-string data type (except Object) to a string.

Normally the Component Processor automatically handles data type conversions. However, for some operations, such as comparisons, you want to specify the data type explicitly. Assume, for example, that you have two fields FIELD_I and FIELD_J containing number values 5000 and 10000. As character fields, 10000 is less than 5000 (because the first character in 10000 is less than the first character in 5000). As numbers, however, 10000 is of course greater than 5000.

Note. Due to the internal representation of numbers, sometimes String represents numbers differently. If you want to control exactly how a number is represented, use the **NumberToString** function.

Parameters

Parameter	Description
value	A value of any type other than object, to be converted to its String representation.

Returns

Returns a String value representing *value*.

Example

To force the comparison of the two fields as strings, you could use:

```
if string(FIELD_1) > string(FIELD_2). . .
```

You can use the String function with a field object as follows:

```
&DATE = GetRecord(RECORD.DERIVED_HR).GetField(FIELD.EFFDT);
&STR = String(&DATE.Value);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Char , page 64, Chapter 1, PeopleCode Built-in Functions, Exact , page 170, Chapter 1, PeopleCode Built-in Functions, Find , page 185, Chapter 1, PeopleCode Built-in Functions, Left , page 313, Chapter 1, PeopleCode Built-in Functions, Substring , page 452, Chapter 1, PeopleCode Built-in Functions, Value , page 493, Chapter 1, PeopleCode Built-in Functions, NumberToString , page 337.

Substitute

Syntax

```
Substitute(source_text, old_text, new_text)
```

Description

Substitute replaces every occurrence of a substring found in a string with a new substring. To replace text that occurs in a specific location in a text string use Replace.

Parameters

Parameter	Description
source_text	A String in which you want to replace substrings.
old_text	A String equal to the substring of *source_text* you want to replace.
new_text	A String with which to replace occurrences of *old_text* in *source_text*.

Returns

Returns a String resulting from replacing every occurrence of *old_text* found in *source_text* with *new_text*.

Example

The following example changes "Second Annual Conference" to "Third Annual Conference":

```
&newstr = Substitute("Second Annual Conference","Second","Third");
```

The next example sets &newstr to "cdcdcd":

```
&newstr = Substitute("ababab", "ab", "cd");
```

See Also

Chapter 1, PeopleCode Built-in Functions, Replace , page 373.

Substring

Syntax

```
Substring(source_str, start_pos, length)
```

Description

Substring extracts a substring of a specified number of characters beginning at a specified location in a source string.

If you know the exact length of *source_str*, and that it is null terminated, you can set *length* to 1 plus the exact length of *source_str* to get everything from *start_pos* to the end.

Parameters

Parameter	Description
source_str	A String from which to extract a substring.
start_pos	A Number representing the character position in *source_str* where the substring starts.
length	A Number specifying the number of characters in the substring.

Returns

Returns a String equal to a substring *length* characters long beginning at character *start* of *source_str*.

Example

This example sets &PAGE_NAME to the first eight characters of the name of the current page:

```
&PAGE_NAME = Substring(%page,1,8);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Char , page 64, Chapter 1, PeopleCode Built-in Functions, Exact , page 170, Chapter 1, PeopleCode Built-in Functions, Find , page 185, Chapter 1, PeopleCode Built-in Functions, Left , page 313, Chapter 1, PeopleCode Built-in Functions, Right , page 381, Chapter 1, PeopleCode Built-in Functions, String , page 451, Chapter 1, PeopleCode Built-in Functions, Substringb , page 453.

Substringb

Syntax

Substringb(*source_str, start_pos, length*)

Description

Substringb extracts a substring of a specified number of bytes beginning at a specified location in a source string. **Substringb** is byte-oriented, whereas **Substring** is character-oriented; this distinction is significant whenever the source string contains double-byte characters.

If you know the exact length of *source_str*, and that it is null terminated, you can set *length* to 1 plus the exact length of *source_str* to get everything from *start_pos* to the end.

Parameters

Parameter	Description
source_str	A String from which to extract a substring.
start_pos	A Number representing the byte position in *source_str* where the substring begins.
length	A Number specifying the number of bytes in the substring.

Returns

Returns a String value equal to a substring *length* bytes long beginning at byte *start* of *source_str*.

Example

This example sets &PAGE_NAME to the first eight bytes of the name of the current page:

```
&PAGE_NAME = Substringb(%page,1,8);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Char , page 64, Chapter 1, PeopleCode Built-in Functions, Exact , page 170, Chapter 1, PeopleCode Built-in Functions, Find , page 185, Chapter 1, PeopleCode Built-in Functions, Left , page 313, Chapter 1, PeopleCode Built-in Functions, Right , page 381, Chapter 1, PeopleCode Built-in Functions, String , page 451, Chapter 1, PeopleCode Built-in Functions, Substring , page 452.

SwitchUser

Syntax

SwitchUser(*UserID, Password, AuthToken , ExtAuthInfo*)

Note. *Password* is *not* encrypted: it is passed as a string.

Description

The **SwitchUser** function changes the user ID of the current user logged onto the PeopleSoft system.

Note. SwitchUser changes only the user for the session of the local application server. It does *not* cascade back to the portal or to any other application servers in the environment.

The SwitchUser function might be used as follows. Suppose there is a special user ID in the system called REGIST. REGIST *only* has access to the self-registration component. The self-registration component has logic that asks the user a list of questions and information based on data in the database. Are you a customer, vendor, or employee? Enter your customer name. Enter other information related to this customer account (such as information only this customer knows or information this customer just received from a workflow email). After the program verifies the information, create a User ID for this customer. After the user ID is created, the program should take the user directly into their transaction without having to logoff, by using SwitchUser.

Note. You must never call SwitchUser from Signon PeopleCode. SwitchUser calls Signon PeopleCode, therefore creating an infinite loop.

Parameters

Parameter	Description
UserID	Specify the User ID to be started. This parameter takes a string value.
Password	Specify the Password for this User ID. This parameter takes a string value.

Note. *Password* is *not* encrypted: it is passed as a string.

Parameter	Description
AuthToken	Specify a single signon authentication token used to authenticate the user. If you are authenticating the user by Userid and password, specify a NULL value for this parameter, that is, two quotation marks with no blank space between them (""). If you specify a token, and the token is valid, SwitchUser switches to the User ID embedded in the token. All other parameters are ignored if a token is used. This parameter takes a string value.
ExtAuthInfo	Specify binary data (encoded as a base64 string) used as additional input to authenticate the user. If your application doesn't use external authentication information, specify a NULL value for this parameter, that is, two quotation marks with no blank space between them ("").

Returns

A Boolean value: True if user ID is switched successfully, False otherwise.

Example

The most common use of SwitchUser specifies only a Userid and Password. If the SwitchUser function executes successfully, you should check to see if the password for the new user id has expired.

```
If Not SwitchUser("MYUSERID", "MYPASSWORD", "", "") Then
   /* switch failed, do error processing */
Else
   If %PasswordExpired Then
      /* application specific processing for expired passwords */
   End-If;
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetPasswordExpired , page 424, Chapter 3, System Variables , %UserId , page 581, Chapter 3, System Variables , %PasswordExpired , page 574.
PeopleTools PeopleBook: Peoplesoft Security

SyncRequestXmlDoc

Syntax

SyncRequestXmlDoc(*&XmlDoc*, **Message.***MessageName* [, **Node.***NodeName*])

Description

Use the **SyncRequestXmlDoc** function to send a synchronous message that is based on an XmlDoc object. The XmlDoc object must already be instantiated and populated. The message included in the function call should be an *unstructured* message, that is, one that isn't based on a hierarchical record structure.

If you want to handle an XmlDoc as a Message object, you need to define a Message object with a hierarchical structure and migrate the data in the XmlDoc object into the Message object.

Parameters

Parameter	Description
&XmlDoc	Specify an already instantiated and populated XmlDoc object that you want to send as a synchronous message.
MessageName	Specify an already existing unstructured message, prefixed with the reserved word **Message**.
NodeName	Specify a node. This is for Sender Specified Routing (SSR) prefixed with the reserved word **Node**. The node defines the target for the published message.

Returns

A reference to an XmlDoc object that is the response.

Example

```
Local XmlDoc &reqdoc, &respdoc;

. . .

&respdoc = SyncRequestXmlDoc(&reqdoc, Message.MY_MESSAGE, Node.MY_NODE);
```

See Also

Chapter 1, PeopleCode Built-in Functions, PublishXmlDoc , page 352, Chapter 17, Message Class , SyncRequest, page 997.
PeopleSoft Integration Broker, Defining Message Channels and Messages

Tan

Syntax

Tan(angle)

Description

Tan calculates the tangent of the given angle (opposite / adjacent).

Parameters

Parameter	Description
angle	A value in radians.

Note. In theory, values of *angle* such that *angle* mod pi = pi/2 are not valid for this function, because inputs approaching such values produce results that tend toward infinity. In practice, however, no computer system can represent such values exactly. Thus, for example, the statement **Tan(Radians(90))** produces a number close to the largest value PeopleCode can represent, rather than an error.

Returns

A real number.

Example

The following example returns the tangent of an angle measuring **1.2** radians:

```
&MY_RESULT = Tan(1.2);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Acos , page 29, Chapter 1, PeopleCode Built-in Functions, Asin , page 45, Chapter 1, PeopleCode Built-in Functions, Atan , page 46, Chapter 1, PeopleCode Built-in Functions, Cos , page 91, Chapter 1, PeopleCode Built-in Functions, Cot , page 92, Chapter 1, PeopleCode Built-in Functions, Degrees , page 134, Chapter 1, PeopleCode Built-in Functions, Radians , page 356, Chapter 1, PeopleCode Built-in Functions, Sin , page 439.

Throw

Syntax

Throw *Expression*

Description

Use the **Throw** statement to throw an exception. This can be used to create your own exceptions, instead of using one generated by the system.

Parameters

Parameter	Description
Expression	Specify the exception object that you want to throw. This can either be an already defined and created exception object, or one that you create with the Throw statement.

Returns

None.

Example

```
Local Exception &ex;

Function t1(&i As integer) Returns number

   Local number &res = &i / 0;

End-Function;

Function t2
   throw CreateException(2, 160, "'%1' doesn't support property or method '%2'",⇒
 "SomeClass", "SomeMethod");
End-Function;

try
   Local number &res = t1(2);
catch Exception &caught
```

```
      MessageBox(0, "", 0, 0, "Caught exception: " | &caught.ToString());
   end-try;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateException , page 97, Chapter 1, PeopleCode Built-in Functions, Try , page 482.
PeopleSoft Integration Broker, Defining Message Channels and Messages

Time

Syntax

Time(*n*)

Description

Time derives a Time value from a Number value. Use it to assign values to Time fields and variables, since Time values cannot be directly represented as constants.

Parameters

Parameter	Description
n	A Number in the form HHMMSS[.SSSSSS], representing a time to a precision of up to .000001 second, based on a 24-hour clock.

Returns

Returns a Time value based on the number *n*.

Example

The example sets &START_TIME to 12:34:56.123456:

```
&START_TIME = Time(123456.123456);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date , page 117, Chapter 1, PeopleCode Built-in Functions, DateTimeValue , page 125, Chapter 1, PeopleCode Built-in Functions, Time3 , page 458, Chapter 1, PeopleCode Built-in Functions, TimeValue , page 461.

Time3

Syntax

Time3(*hours*, *mins*, *secs*)

Description

Time3 derives a Time value from 3 supplied numbers. It can be used to assign values to Time fields and variables, since Time values cannot be directly represented as constants.

Parameters

Parameter	Description
hours	A Number in the form HH between 00 and 23, representing hours on a 24-hour clock.
mins	A Number in the form MM between 00 and 59, representing minutes of the hour.
secs	A Number in the form SS[.SSSSSS], between 00 and 59.999999, representing seconds.

Returns

Returns a Time value based equal to the sum of the three input values representing hours, minutes, and seconds, to a precision of .000001 second.

Example

The example sets &START_TIME to 11.14.09.300000:

```
&START_TIME = Time3(11,14,9.3);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date3 , page 118, Chapter 1, PeopleCode Built-in Functions, DateTime6 , page 119, Chapter 1, PeopleCode Built-in Functions, Time , page 458, Chapter 1, PeopleCode Built-in Functions, TimeValue , page 461.

TimePart

Syntax

TimePart(*datetime_val*)

Description

TimePart derives the Time component of a DateTime value.

Parameters

Parameter	Description
datetime_val	A DateTime value from which to extract the Time component.

Returns

Returns a Time value equal to the time portion of *datetime_val*.

Example

The example set &T to 15.34.35.000000:

```
&DT = DateTimeValue("12/13/93 3:34:35 PM");
&T = TimePart(&DT);
```

See Also

Chapter 1, PeopleCode Built-in Functions, DatePart , page 118, Chapter 1, PeopleCode Built-in Functions, Hour , page 291, Chapter 1, PeopleCode Built-in Functions, Minute , page 326, Chapter 1, PeopleCode Built-in Functions, Second , page 393.

TimeToTimeZone

Syntax

```
TimeToTimeZone(OldTime, SourceTimeZone, DestinationTimeZone);
```

Description

TimeToTimeZone converts a time field from the time specified by *SourceTimeZone* to the time specified by *DestinationTimeZone*.

Considerations Using this Function

This function should generally be used in PeopleCode, *not* for displaying time. If you take a time value, convert it from base time to client time, then try to display this time, depending on the user settings, when the time is displayed the system might try to do a *second* conversion on an already converted time. This function could be used as follows: suppose a user wanted to check to make sure a time was in a range of times on a certain day, in a certain timezone. If the times were between 12 AM and 12PM in EST, these resolve to 9 PM and 9AM PST, respectively. The start value is *after* the end value, which makes it difficult to make a comparison. This function could be used to do the conversion for the comparison, in temporary fields, and not displayed at all.

Parameters

Parameter	Description
OldTime	Specify the time value to be converted.
SourceTimeZone	Specify the time zone that *OldTime* is in. Values are:
	timezone - a time zone abbreviation or a field reference to be used for converting *OldTime*.
	Local - use the local time zone for converting *OldTime*.
	Base - use the base time zone for converting *OldTime*.
DestinationTimeZone	Specify the time zone that you want to convert *OldTime* to. Values are:
	timezone - a time zone abbreviation or a field reference to be used for converting *OldTime*.
	Local - use the local time zone for converting *OldTime*.
	Base - use the base time zone for converting *OldTime*.

Returns

A converted time value.

Example

The following example TESTTM is a time field with a value 01/01/99 10:00:00. This example converts TESTTM from Eastern Standard Time (EST) to Pacific Standard Time (PST).

```
&NEWTIME = TimeToTimeZone(TESTTM, "EST", "PST");
```

&NEWTIME is a time variable with a value of 7:00:00AM.

See Also

Chapter 1, PeopleCode Built-in Functions, ConvertDatetimeToBase , page 85, Chapter 1, PeopleCode Built-in Functions, ConvertTimeToBase , page 87, Chapter 1, PeopleCode Built-in Functions, FormatDateTime , page 192, Chapter 1, PeopleCode Built-in Functions, IsDaylightSavings , page 302, Chapter 1, PeopleCode Built-in Functions, DateTimeToTimeZone , page 124, Chapter 1, PeopleCode Built-in Functions, TimeZoneOffset , page 461.
PeopleSoft Server Tools Administration, PeopleTools Utilities

TimeValue

Syntax

TimeValue(*time_str*)

Description

TimeValue calculates a Time value based on an input string. This function can be used to assign a value to a Time variable or field using a string constant, since a Time value cannot be represented with a constant.

Parameters

Parameter	Description
time_str	A String representing a time in the form HH:MM:SS.SSSSSS, based on a 24-hour clock.

Returns

Returns a Time value based on *time_str*.

Example

The example sets &START_TIME to 12.13.00.000000:

```
&START_TIME = TimeValue("12:13:00 PM");
```

See Also

Chapter 1, PeopleCode Built-in Functions, DateTimeValue , page 125, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126.

TimeZoneOffset

Syntax

TimeZoneOffset(*DateTime* {[, *timezone* | **"Base"** | **"Local"**]})

Description

The **TimeZoneOffset** function generates a time offset for *datetime*. The offset represents the relative time difference to GMT. If no other parameters are specified with *datetime*, the server's base time zone is used.

Parameters

Parameter	Description
datetime	Specify the datetime value to be used for generating the offset.
timezone | **Local** | **Base**	Specify a value to be used with *datetime*. The values are:
	timezone - a time zone abbreviation or a field reference to be used with *datetime*.
	Local - use the local time zone with *datetime*.
	Base - use the base time zone with *datetime*.

Returns

An offset string of the following format:

```
Shh:mm
```

where

S	is + or -, with + meaning East of Greenwich
hh	is the hours of offset
mm	is the minutes of offset

See Also

Chapter 1, PeopleCode Built-in Functions, ConvertDatetimeToBase , page 85, Chapter 1, PeopleCode Built-in Functions, ConvertTimeToBase , page 87, Chapter 1, PeopleCode Built-in Functions, FormatDateTime , page 192, Chapter 1, PeopleCode Built-in Functions, IsDaylightSavings , page 302, Chapter 1, PeopleCode Built-in Functions, DateTimeToTimeZone , page 124, Chapter 1, PeopleCode Built-in Functions, TimeToTimeZone , page 460.

TotalRowCount

Syntax

TotalRowCount(*scrollpath*)

Where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,] **RECORD.***target_recname*

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

TotalRowCount calculates the number of rows (including rows marked as deleted) in a specified scroll area of a page.

Note. This function remains for backward compatibility only. Use the **RowCount** Rowset property instead.

Rows that have been marked as deleted remain accessible to PeopleCode until the database has been updated; that is, all the way through SavePostChange.

TotalRowCount is used to calculate the upper limit of a **For** loop if you want the loop to go through rows in the scroll that have been marked as deleted. If the logic of the loop does not need to execute on deleted rows, use **ActiveRowCount** instead.

See Also

Chapter 1, PeopleCode Built-in Functions, ActiveRowCount , page 29, Chapter 1, PeopleCode Built-in Functions, For , page 192, Chapter 27, Rowset Class , RowCount, page 1508, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.

Returns

Returns a Number equal to the total rows (including rows marked as deleted) in the target scroll.

Example

The example uses **TotalRowCount** to calculate the limiting value on a **For** loop, which loops through all the rows in the scroll area:

```
&ROW_COUNT = TotalRowCount(RECORD.BUS_EXPENSE_PER, CurrentRowNumber(1), ⇒
RECORD.BUS_EXPENSE_DTL);
for &I = 1 to &ROW_COUNT
 /* do something with row &I that has to be done to deleted as well as active rows⇒
 */
end-for;
```

See Also

Chapter 1, PeopleCode Built-in Functions, ActiveRowCount , page 29, Chapter 1, PeopleCode Built-in Functions, CopyRow , page 90, Chapter 1, PeopleCode Built-in Functions, CurrentRowNumber , page 116, Chapter 1, PeopleCode Built-in Functions, FetchValue , page 181, Chapter 1, PeopleCode Built-in Functions, For , page 192.

Transfer

Syntax

Transfer(*new_instance*,

```
MENUNAME.menuname,
BARNAME.barname,
ITEMNAME.menu_itemname,
PAGE.component_item_name,
action [, keylist] [, AutoSearch]);
```

where *keylist* is a list of field references in the form:
`[recordname.]field1 [, [recordname.]field2]. . .`

OR
`&RecordObject1 [, &RecordObject2]. . .`

Description

Transfer closes the current page and transfers the end-user to another page, either within the current component or in another component. **Transfer** can either start a new instance of the application and transfer to the new page there, or close the old page and transfer to the new one in the same instance of PeopleTools.

Transfer is more powerful than the simpler TransferPage, which permits a transfer only within the current component in the current instance of PeopleTools. However, any variables declared as Component do *not* remain defined after using the **Transfer** function, whether you're transferring within the same component or not.

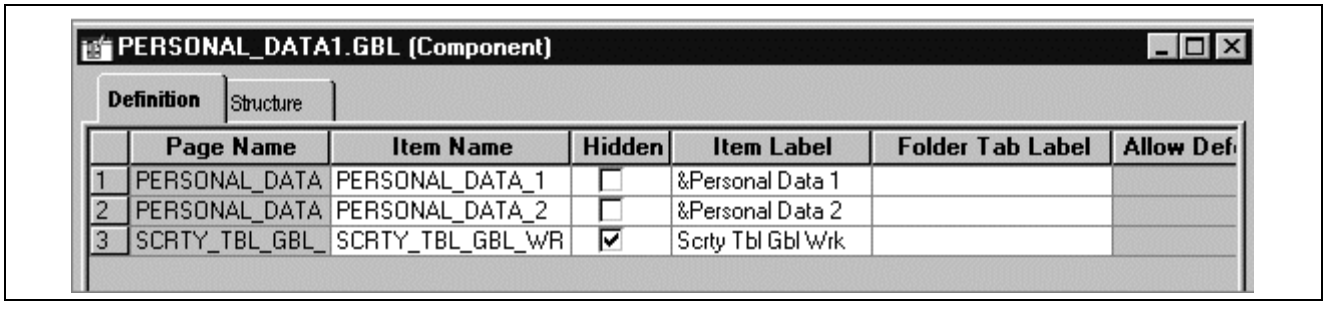
You can use **Transfer** from a secondary page (either with or without using a pop-up menu) *only* if you're transferring to a separate instance of a component. You *cannot* use **Transfer** from a secondary page if you're not transferring to a separate instance of a component.

If you provide a valid search key for the new page in the optional *keylist*, the new page opens directly, using the values provided from *keylist* as search key values. A valid key means that enough information is provided to uniquely identify a row: not all of the key values need to be provided. If no key is provided, or if the key is invalid, or if not enough information is provided to identify a unique row, the search dialog box displays, enabling the end user to search for a row.

Note. If Force Search Processing is specified in PeopleSoft Application Designer for the component, the search dialog box always displays, whether the keylist is provided or not.

If *barname+itemname+component_item_name* is an invalid combination, an error message displays explaining that there were invalid transfer parameters.

In the *component_item_name* parameter, make sure to pass the component item name for the page, not the page name. The component item name is specified in the Component Definition, in the Item Name column on the row corresponding to the specific page, as shown here:



Component Item Name

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Restrictions on Use With SearchInit Event

You can't use this function in a SearchInit PeopleCode program.

Considerations Using Exceptions and the Transfer Function

Using the **Transfer** function, inside a try block, to transfer a user to a page in another component, does *not* catch PeopleCode exceptions thrown in the new component. Starting a new component starts a brand new PeopleCode evaluation context. Catches are only caught for exceptions thrown within the *current* component.

In the following code example, the catch statement only catches exceptions thrown in the code *prior to* using the DoModal function, but not any exceptions that are thrown within the new component.

```
/* Set up transaction */
If %CompIntfcName = "" Then
   try
      &oTrans = &g_ERMS_TransactionCollection.GetTransactionByName(RB_EM_⇒
WRK.DESCR);
      &sSearchPage = &oTrans.SearchPage;
      &sSearchRecord = &oTrans.SearchRecord;
      &sSearchTitle = &oTrans.GetSearchPageTitle();
      If Not All(&sSearchPage, &sSearchRecord, &sSearchTitle) Then
         Error (MsgGetText(17834, 8081, "Message Not Found"));
      End-If;
      &c_ERMS_SearchTransaction = &oTrans;

      /* Attempt to transfer to hidden search page with configurable filter */
      &nModalReturn = DoModal(@("Page." | &sSearchPage), &sSearchTitle, - 1, - 1);
   catch Exception &e
      Error (MsgGetText(17834, 8082, "Message Not Found"));
   end-try;
```

See Also

Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475, *PeopleSoft Application Designer*, Creating Menu Definitions, *PeopleSoft Application Designer*, Creating Component Definitions.

Parameters

Parameter	Description
new_instance	Set this parameter to True to start a new application instance, or to False to use the current window and replace the page already displayed.
Menuname	The name of the menu where the page is located prefixed with the reserved word **MENUNAME**.
Barname	The name of the menu bar where the page is located, prefixed with the reserved word **BARNAME**.
menu_itemname	The name of the menu item where the page is located, prefixed with the reserved word **ITEMNAME**.
component_item_name	The component item name of the page to be displayed on top of the component when it displays. The component item name is specified in the component definition. This parameter must be prefixed with the keyword **PAGE**.
Action	Uses a single-character code as in %Action. Valid actions are "A" (add), "U" (update), "L" (update/display all), "C" (correction), and "E" (data entry).
Keylist	An optional list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. If a record object is specified, any field of that record object *that is also a field of the search record* for the destination component is added to keylist. The keys in the *fieldlist* must uniquely identify a row in the "to" page search record. If a unique row is not identified, or if Force Search Processing is selected for the component, the search dialog box appears. If the *keylist* parameter is not supplied then the destination component's search key must be found as part of the source components level 0 record buffer.
AutoSearch	Specify whether an automatic search on the target search page is executed after the transfer. This means the search results are already shown without the end-user having to click the Search button. This parameter takes a Boolean value: True, do an automatic search. The default value is False (that is, the user has to click the Search button).

Returns

None.

Example

The example starts a new instance of PeopleTools and transfers to a new page in Update mode. The data in the new page is selected by matching the EMPLID field from the old page.

```
Transfer(true, MENUNAME.ADMINISTER_PERSONNEL, BARNAME.USE, ITEMNAME.  ⇒
PERSONAL_DATA, PAGE.PERSONAL_DATA_1, "U");
```

The following example is used with workflow.

```
Local Record &WF_WL_DEFN_VW, &MYREC, &PSSTEPDEFN;

If All(WF_WORKLIST_VW.BUSPROCNAME) Then
```

```
  &BPNAME = FetchValue(WF_WORKLIST_VW.BUSPROCNAME, CurrentRowNumber());
  &WLNAME = FetchValue(WF_WORKLIST_VW.WORKLISTNAME, CurrentRowNumber());
  &INSTANCEID = FetchValue(WF_WORKLIST_VW.INSTANCEID, CurrentRowNumber());

  &WF_WL_DEFN_VW = CreateRecord(RECORD.WF_WL_DEFN_VW);
  &PSSTEPDEFN = CreateRecord(RECORD.PSSTEPDEFN);

  SQLExec("select %List(SELECT_LIST, :1) from %Table(:1) where Busprocname = :2⇒
and Worklistname = :3", &WF_WL_DEFN_VW, &BPNAME, &WLNAME, &WF_WL_DEFN_VW);

  SQLExec("select %List(SELECT_LIST, :1) from %Table(:1) where Activityname = :2⇒
and Stepno = 1 and Pathno = 1", &PSSTEPDEFN, &WF_WL_DEFN_VW.ACTIVITYNAME.Value,⇒
&PSSTEPDEFN);

  Evaluate &PSSTEPDEFN.DFLTACTION.Value
  When = 0
     &ACTION = "A";
  When = 1
     &ACTION = "U";
  When-Other
     &ACTION = "U";
  End-Evaluate;

  &MYREC = CreateRecord(@("RECORD." | &WF_WL_DEFN_VW.WLRECNAME.Value));

  SQLExec("Select %List(SELECT_LIST, :1) from %Table(:1) where Busprocname = :2⇒
and Worklistname = :3 and Instanceid = :4", &MYREC, &BPNAME, &WLNAME,⇒
&INSTANCEID, &MYREC);

  Transfer( True, @("MENUNAME." | &PSSTEPDEFN.MENUNAME.Value), @("BARNAME." |⇒
 &PSSTEPDEFN.BARNAME.Value), @("ITEMNAME." | &PSSTEPDEFN.ITEMNAME.Value), @⇒
("PAGE." | &PSSTEPDEFN.PAGEITEMNAME.Value), &ACTION, &MYREC);

  End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475, Chapter 1, PeopleCode Built-in Functions, DoModalComponent , page 150.

TransferMobilePage

Syntax

```
TransferMobilePage([MOBILEPAGE.]PageName,
Tab, &CIObject);
```

Description

The **TransferMobilePage** function closes the current Mobile Page Detail View and transfers the end user to another Mobile Page Detail View. This function is only supported in the OnChange event or on the OnInit event for level 0 objects.

A new working set is *not* started when transferring between objects in the same component.

The *Tab* parameter takes the definitional number, not the number of the tab that is displaying. For example, if there are three tabs defined, but only tabs one and three are showing (you've hidden tab number two) you would specify a 3 for the *Tab* parameter to transfer to tab number three, even though it is displaying as tab number two.

Note. When a user is transferred using TransferMobilePage, the Return To list hyperlink is not displayed on the Mobile Page.

Parameters

Parameter	Description
PageName	Specify the mobile page definition used to display the page. This parameter takes a string value. It is good programming practice to preface this parameter with the keyword **MOBILEPAGE**, since it provides automatic rename support, however, it is not necessary to use it.
Tab	Specify an integer (1 to *n*) that determines what tab to make active. This parameter takes a number value.
&CIObject	Specify an already instantiated mobile object.

Returns

None.

Example

```
&MyCI = %Session.GetCompIntfc(COMPINTFC.myci);

&MyCI.Key1 = "x";
&MyCI.Ket2 = "y";

If Not(&MyCI.Get()) Then;
     errorstuff();
End-if;

Transfer(MOBILEPAGE.MyCIPageName, 1, &MyCI);
```

TransferNode

Syntax

```
TransferNode(new_instance,
      NODE.nodename,
      MENUNAME.menuname,
      MARKET.marketname,
```

```
COMPONENT.componentname,
PAGE.component_item_name,
action [, keylist]);
```

where *keylist* is a list of field references in the form:
`[recordname.]field1 [, [recordname.]field2]. . .`

OR
`&RecordObject1 [, &RecordObject2]. . .`

Description

The **TransferNode** function is used to transfer the user to a page in another Node, but within the same portal.

TransferNode can either start a new instance of the application and transfer to the new page, or close the old page and transfer to the new one in the same instance of PeopleTools.

Component scoped and Global scoped variables are not available if the new page is in a different node.

Entering null values ("") for the node opens the new component within the current node or portal.

If you want to transfer the end-user to another portal, use the TransferPortal function.

If you provide a valid search key for the new page in the optional *fieldlist*, the new page opens directly, using the values provided from *fieldlist* as search key values. If no key is provided, or if the key is invalid, the search dialog displays, allowing the end-user to search for a row.

Note. If Force Search Processing is specified in Application Designer for the component, the search dialog always displays, whether the keylist is provided or not.

If **TransferNode** is called in a RowInit PeopleCode program, the PeopleCode program is terminated. *However*, the component processor continues with its RowInit processing, calling RowInit on the other fields. The actual transfer won't happen until after that completes. You may want to place any TransferPage functions in the Activate event for the page, or later in the Component Processor event flow.

See *PeopleSoft Application Designer*, Creating Component Definitions.

Restrictions on Use with a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Restrictions on Use with SearchInit Event

You can't use this function in a SearchInit PeopleCode program.

Considerations Using Exceptions and the TransferNode Function

Using the **TransferNode** function inside a try blockdoes *not* catch PeopleCode exceptions thrown in the new component. Starting a new component starts a brand new PeopleCode evaluation context. Catches are only caught for exceptions thrown within the *current* component.

In the following code example, the catch statement only catches exceptions thrown in the code *prior to* using the DoModal function, but not any exceptions that are thrown within the new component.

```
/* Set up transaction */
If %CompIntfcName = "" Then
   try
      &oTrans = &g_ERMS_TransactionCollection.GetTransactionByName(RB_EM_⇒
WRK.DESCR);
      &sSearchPage = &oTrans.SearchPage;
      &sSearchRecord = &oTrans.SearchRecord;
      &sSearchTitle = &oTrans.GetSearchPageTitle();
      If Not All(&sSearchPage, &sSearchRecord, &sSearchTitle) Then
         Error (MsgGetText(17834, 8081, "Message Not Found"));
      End-If;
      &c_ERMS_SearchTransaction = &oTrans;

      /* Attempt to transfer to hidden search page with configurable filter */
      &nModalReturn = DoModal(@("Page." | &sSearchPage), &sSearchTitle, - 1, - 1);
   catch Exception &e
      Error (MsgGetText(17834, 8082, "Message Not Found"));
   end-try;
```

Parameters

Parameter	Description
new_instance	Set this parameter to True to start a new application instance, or to False to use the current window and replace the page already displayed.
nodename	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
menuname	Specify the name of the menu containing the content, prefixed with the reserved word **MENUNAME**. You can also use a string, such as %Menu, for this value.
marketname	Specify the name of the market of the component, prefixed with the reserved word **MARKET**. You can also use a string, such as %Market, for this value.
component_item_name	Specify the component item name of the page to be displayed on top of the component when it displays. The component item name is specified in the component definition. If you specify a page, it must be prefixed with the keyword **PAGE**. You can also specify a null ("") for this parameter.
action	Specify a single-character code. Valid actions are: • "A" (add) • "U" (update) • "L" (update/display all) • "C" (correction) • "E" (data entry) You can also specify a null ("") for this parameter.
keylist	An optional list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. If a record object is specified, any field of that record object *that is also a field of the search record for the destination component* is added to keylist. The keys in the *fieldlist* must uniquely identify a row in the "to" page search record. If a unique row is not identified, of if Force Search Processing has been selected, the search dialog appears. If the *keylist* parameter is not supplied then the destination component's search key must be found as part of the source components level 0 record buffer.

Returns

A Boolean value: True if function completed successfully, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, TransferPortal , page 471, Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475, Chapter 1, PeopleCode Built-in Functions, Transfer , page 463.

TransferPortal

Syntax

```
TransferPortal(new_instance,
      PORTAL.portalname,
```

```
NODE.nodename,
MENUNAME.menuname,
MARKET.marketname,
COMPONENT.componentname,
PAGE.component_item_name,
action [, keylist]);
```

where *keylist* is a list of field references in the form:
`[recordname.]field1 [, [recordname.]field2]. . .`

OR
`&RecordObject1 [, &RecordObject2]. . .`

Description

The **TransferPortal** function is used to transfer the user to a page in another Node in a different portal.

TransferPortal can either start a new instance of the application and transfer to the new page, or close the old page and transfer to the new one in the same instance of PeopleTools.

Component scoped and Global scoped variables are not available after this function.

If you want to transfer the end-user to another node within the same portal, use the TransferNode function.

If you provide a valid search key for the new page in the optional *fieldlist*, the new page opens directly, using the values provided from *fieldlist* as search key values. If no key is provided, or if the key is invalid, the search dialog displays, allowing the end-user to search for a row.

Note. If Force Search Processing is specified in Application Designer for the component, the search dialog always displays, whether the keylist is provided or not.

If **TransferPortal** is called in a RowInit PeopleCode program, the PeopleCode program is terminated. *However*, the component processor continues with its RowInit processing, calling RowInit on the other fields. The actual transfer won't happen until after that completes. You may want to place any TransferPortal functions in the Activate event for the page, or later in the Component Processor flow.

See *PeopleSoft Application Designer*, Creating Component Definitions.

Restrictions on Use with a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Restrictions on Use with SearchInit Event

You can't use this function in a SearchInit PeopleCode program.

Restrictions on Use with Different Releases

You cannot use this function to transfer a user from a PeopleTools 8.42 portal to any base PeopleTools 8.1x portal that overwrites the expired cookie value when login occurs.

The TransferPortal function is currently supported to transfer users to pages in other nodes to base PeopleTools 8.18 portals, including all PeopleTools 8.18 versions and patches.

Considerations Using Exceptions and the TransferPortal Function

Using the **TransferPortal** function inside a try block does *not* catch PeopleCode exceptions thrown in the new component. Starting a new component starts a brand new PeopleCode evaluation context. Catches are only caught for exceptions thrown within the *current* component.

In the following code example, the catch statement only catches exceptions thrown in the code *prior to* using the DoModal function, but not any exceptions that are thrown within the new component.

```
/* Set up transaction */
If %CompIntfcName = "" Then
   try
      &oTrans = &g_ERMS_TransactionCollection.GetTransactionByName(RB_EM_⇒
WRK.DESCR);
      &sSearchPage = &oTrans.SearchPage;
      &sSearchRecord = &oTrans.SearchRecord;
      &sSearchTitle = &oTrans.GetSearchPageTitle();
      If Not All(&sSearchPage, &sSearchRecord, &sSearchTitle) Then
         Error (MsgGetText(17834, 8081, "Message Not Found"));
      End-If;
      &c_ERMS_SearchTransaction = &oTrans;

      /* Attempt to transfer to hidden search page with configurable filter */
      &nModalReturn = DoModal(@("Page." | &sSearchPage), &sSearchTitle, - 1, - 1);
   catch Exception &e
      Error (MsgGetText(17834, 8082, "Message Not Found"));
   end-try;
```

Parameters

Parameter	Description
new_instance	Set this parameter to True to start a new application instance, or to False to use the current window and replace the page already displayed.
PortalName	Specify the name of the portal that you want to transfer to, prefixed with the reserved word **Portal**.
nodename	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
menuname	Specify the name of the menu containing the content, prefixed with the reserved word **MENUNAME**. You can also use a string, such as %Menu, for this value.
marketname	Specify the name of the market of the component, prefixed with the reserved word **MARKET**. You can also use a string, such as %Market, for this value.
component_item_name	Specify the component item name of the page to be displayed on top of the component when it displays. The component item name is specified in the component definition. If you specify a page, it must be prefixed with the keyword **PAGE**. You can also specify a null ("") for this parameter.
action	Specify a single-character code. Valid actions are: • "A" (add) • "U" (update) • "L" (update/display all) • "C" (correction) • "E" (data entry) You can also specify a null ("") for this parameter.
keylist	An optional list of field specifications used to select a unique row at level zero in the page you are transferring to, by matching keys in the page you are transferring from. It can also be an already instantiated record object. If a record object is specified, any field of that record object *that is also a field of the search record for the destination component* is added to keylist. The keys in the *fieldlist* must uniquely identify a row in the "to" page search record. If a unique row is not identified, of if Force Search Processing has been selected, the search dialog appears. If the *keylist* parameter is not supplied then the destination component's search key must be found as part of the source components level 0 record buffer.

Returns

A Boolean value: True if function completed successfully, False otherwise.

See Also

Chapter 1, PeopleCode Built-in Functions, TransferNode , page 468, Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475, Chapter 1, PeopleCode Built-in Functions, Transfer , page 463.

TransferPanel

Syntax

```
TransferPanel([PANEL.panel_name])
```

Description

TransferPanel transfers control to the panel indicated by **PANEL.** *panel_name* within, or to the panel set with the **SetNextPage** function.

Note. The **TransferPanel** function is supported for compatibility with previous releases of PeopleTools. New applications should use **TransferPage** instead.

See Also

Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475.

TransferPage

Syntax

```
TransferPage([PAGE.page_name])
```

Description

TransferPage transfers control to the page indicated by **PAGE.** *page_name* within, or to the page set with the **SetNextPage** function. The page that you transfer to *must* be in the current component or menu. To transfer to a page outside the current component or menu, or to start a separate instance of PeopleTools prior to transfer into, use the **Transfer** function.

See Chapter 1, PeopleCode Built-in Functions, SetNextPage , page 422 and Chapter 1, PeopleCode Built-in Functions, Transfer , page 463.

Note. You can't use **TransferPage** from a secondary page.

Any variable declared as a Component variable will still be defined after using a **TransferPage** function.

Considerations Using TransferPage

The following are important considerations when using the **TransferPage** function:

- TransferPage always terminates the current PeopleCode program.
- TransferPage is always processed after all events are completed.

Given these considerations, here are some scenarios for how TransferPage executes:

- When called in RowInit: The current RowInit PeopleCode program is terminated, but RowInit processing continues. In addition, RowInit PeopleCode programs run for the rest of the fields in the row. Then TransferPage is processed.
- When called in FieldEdit: The FieldEdit PeopleCode program is terminated. The FieldChange program for that field still runs. Then TransferPage is processed.

- When called in SavePreChange: The SavePreChange program for that field is terminated. SavePreChange runs for the rest of the fields on that page. Then SavePostChange run for all the fields. Then TransferPage is processed.
- When called in FieldChange in deferred mode: In deferred mode, changed fields are processed in order. The FieldChange program is terminated. Then any subsequent fields in the page order are processed with the normal FieldEdit-Field Change logic. Once that has finished, the TransferPage is processed.

When TransferPage is processed, any PeopleCode associated with the **Activate** event for the page being transferred to runs. This always occurs at the end, after all field processing.

If TransferPage is called multiple times during field processing, all the calls are processed at the end, in the same order the calls were made. The Activate event executes each time. The final active page is the one that was transferred to by the last call.

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Restrictions on Use With SearchInit Event

You can't use this function in a SearchInit PeopleCode program.

Parameters

Parameter	Description
page_name	A String equal to the name of the page you are transferring to, as set in the page definition, prefixed with the reserved word **Page**. The page *must* be in the same component as the page you are transferring from.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

The following examples both perform the same function, which is to transfer to the JOB_DATA_4 page:

```
TransferPage(PAGE.JOB_DATA_4);
```

or

```
SetNextPage(PAGE.JOB_DATA_4);
TransferPage( );
```

See Also

Chapter 1, PeopleCode Built-in Functions, DoModalComponent , page 150, Chapter 1, PeopleCode Built-in Functions, SetNextPage , page 422, Chapter 1, PeopleCode Built-in Functions, Transfer , page 463.

Transform

Syntax

```
Transform({XmlString | &XmlDoc}AE_Program_Name,
```

```
Initial_Node_Name,
Initial_Message__Name,
Initial_Message_Version,
Result_Node_Name,
Result_Message_Name,
Result_Message_Version)
```

Description

Use the **Transform** function to modify one transaction, as specified by the *Initial* parameters, to another transaction, specified by the *Result* parameters, using an Application Engine program. This is used with Integration Broker.

Generally using this function implies that you're transforming a message that you're not actually sending or receiving at the current time. By using this method, and specifying the two transactions, it's as if you're defining a relationship, without having to use the relationship component.

Considerations Using the Transform Functions

The **Transform** function uses an existing Application Engine program to do transformations. This enables you to break up the flow of Integration Broker and do transformations when you need to. If you wish to resuse your Application Engine programs, you can invoke them by using this function.

The **TransformEx** function does not use an Application Engine program to do a transformation. Instead, it does an Extensible Stylesheet Language Transformation (XSLT.) This enables you to dynamically do transformations outside of Integration Broker, such as, performing transformations on pagelets in a portal every time a page is accessed.

The **TransformExCache** function also does XSLT transformations without using an Application Engine program, outside of Integration Broker. Use TransformExCache when you have a large volume of similar transformations to be done. Caching technology is used with this function. You may see an increase in performance, as well as an increase in memory consumption, using this function.

Parameters

Parameter	Description
XmlString | *&XmlDoc*	Specify an already populated XmlDoc object, an XML string, or other text that you want transformed.
AE_Program_Name	Specify the name of the Application Engine program that you want to use for the transformation.
Initial_Node_Name	Specify the name of the initial node as a string.
Initial_Message_Name	Specify the name of the initial message.
Initial_Message_Version	Specify the version of the initial message that you want to use.
Result_Node_Name	Specify the result, where you want the transformed message to go to.
Result_Message_Name	Specify the name of the result message, the one to use for the output.
Result_Message_Version	Specify the version of the result message to be used.

Returns

An XmlDoc object containing the resulting XML from the transformation. Null is never returned. If you do not want to display an error to the user, place this function inside a try-catch statement.

See Also

PeopleSoft Integration Broker, Applying Filtering, Transformation, and Translation.

TransformEx

Syntax

TransformEx(*XmlString*, *XsltString*)

Description

Use **TransformEx** to do an XSLT transformation of the specified XML string.

This function also strips off any encoding information located within the XML Declaration.

The input, output, and XSL string must all be well-formed XML. If the output is HTML, it is actually XHTML (which is well-formed XML.)

Considerations Using the Transform Functions

The **Transform** function uses an existing Application Engine program to do transformations. This enables you to break up the flow of Integration Broker and do transformations when you need to. If you wish to resuse your Application Engine programs, you can invoke them by using this function.

The **TransformEx** function does not use an Application Engine program to do a transformation. Instead, it does an Extensible Stylesheet Language Transformation (XSLT.) This enables you to dynamically do transformations outside of Integration Broker, such as, performing transformations on pagelets in a portal every time a page is accessed.

The **TransformExCache** function also does XSLT transformations without using an Application Engine program, outside of Integration Broker. Use TransformExCache when you have a large volume of similar transformations to be done. Caching technology is used with this function. You may see an increase in performance, as well as an increase in memory consumption, using this function.

Parameters

Parameter	Description
XmlString	Specify the XML string that you want transformed.
XsltString	Specify the XSLT string you wish to use to transform the XML string.

Returns

The output of the transformation as a string if successful, NULL otherwise.

Example

```
try
   &outStr = TransformEx(&inXML, &inXSLT);
```

```
catch Exception &E
   MessageBox(0, "", 0, 0, "Caught exception: " | &E.ToString());
end-try;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Transform , page 476.
PeopleSoft Integration Broker, Applying Filtering, Transformation, and Translation

TransformExCache

Syntax

TransformExCache(*&XmlDoc*, *FilePath*, *XsltKey*)

Description

Use **TransformExCache** to do an Extensible Stylesheet Language Transformation (XSLT) transformation of the specified XmlDoc object.

The file specified by *FilePath* must be in well-formed XML.

Considerations Using the Transform Functions

The **Transform** function uses an existing Application Engine program to do transformations. This enables you to break up the flow of Integration Broker and do transformations when you need to. If you wish to resuse your Application Engine programs, you can invoke them by using this function.

The **TransformEx** function does not use an Application Engine program to do a transformation. Instead, it does an Extensible Stylesheet Language Transformation (XSLT.) This enables you to dynamically do transformations outside of Integration Broker, such as, performing transformations on pagelets in a portal every time a page is accessed.

The **TransformExCache** function also does XSLT transformations without using an Application Engine program, outside of Integration Broker. Use TransformExCache when you have a large volume of similar transformations to be done. Caching technology is used with this function. You may see an increase in performance, as well as an increase in memory consumption, using this function.

Parameters

Parameter	Description
&XmlDoc	Specify an already instantiated and populated XmlDoc object that you want transformed.
FilePath	Specify an XSLT file. You must specify an absolute path to the file, including the file extension.
XsltKey	Specify a key to uniquely name the compiled and cached XSLT in the data buffers. This key is used both to create the item in memory as well as retrieve it. This parameter takes a string value, up to 30 characters.

Returns

An XmlDoc object containing the resulting XML from the transformation. Null is never returned. If you do not want to display an error to the user, place this function inside a try-catch statement.

Example

```
Local XmlDoc &inXMLdoc = CreateXmlDoc("");

Local Boolean &ret = &inXMLdoc.ParseXmlFromURL("c:\temp\in.xml");

Local XmlDoc &outDoc = TransformExCache(&inXMLdoc, "c:\temp\in.xsl", "INBOUND");
```

TreeDetailInNode

Syntax

TreeDetailInNode(*setID*, *tree*, *effdt*, *detail_value*, *node*)

Description

TreeDetailInNode determines whether a specific record field value is a descendant of a specified node in a specified tree.

Parameters

Parameter	Description
setID	SetID for the appropriate business unit. This parameter is required. If there is no SetID, you can pass a NULL string ("", not a blank) and a blank will be used.
tree	The tree name that contains the *detail_value*.
effdt	Effective date to be used in the search. You must use a valid date.
detail_value	The *recordname.fieldname* containing the value you are looking for.
node	The "owning" tree node name.

Returns

Returns a Boolean value, True if *detail_value* is a descendant of *node* in *tree*.

Example

This example sets the value of &APPR_RULE_SET to the value at the APPR_RULE_LN record and APPR_RULE_SET fieldname, on the tree ACCOUNT.

```
&APPR_RULE_SET = TreeDetailInNode("SALES", "ACCOUNT", %Date, APPR_RULE_LN.APPR_⇒
RULE_SET, "test");
```

See Also

Chapter 33, Tree Classes , page 1653.
PeopleTools PeopleBook: Peoplesoft Tree Manager

TriggerBusinessEvent

Syntax

```
TriggerBusinessEvent(BUSPROCESS.bus_proc_name,
          BUSACTIVITY.activity_name,
                BUSEVENT.bus_event_name)
```

Description

TriggerBusinessEvent triggers a business event and the workflow routings associated with that event. This function should only be used in Workflow PeopleCode. You can edit Workflow PeopleCode via the Event Definition dialog while you are defining a workflow event.

Parameters

Parameter	Description
bus_proc_name	A string consisting of the name of the business process, as defined in the Business Process Designer, prefixed with the reserved word **BUSPROCESS.**
activity_name	A string consisting of the name of the business activity, as defined in the Business Process Designer, prefixed with the reserved word BUSACTIVITY.
bus_event_name	A string consisting of the name of the business event, as defined in the Business Process Designer, prefixed with the reserved word BUSEVENT.

Returns

Returns a Boolean value: True if successful, false otherwise. The return value is not optional.

Note. You must check the return from TriggerBusinessEvent to see if you have an error. If you have an error, all of the updates up to that TriggerBusinessEvent process are rolled back. However, if you don't halt execution, even if you have an error, all updates *after* the TriggerBusinessEvent process *are* committed. This could result in your database information being out of synch.

Example

The following example triggers the Deny Purchase Request event in the Manager Approval activity of the Purchase Requisition business process:

```
&SUCCESS = TriggerBusinessEvent(BUSPROCESS."Purchase Requisition",⇒
 BUSACTIVITY."Manager Approval", BUSEVENT."Deny Purchase Request");
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetWLFieldValue , page 281, Chapter 1, PeopleCode Built-in Functions, MarkWLItemWorked , page 318, *PeopleSoft Workflow*, Adding Events and Routings, *PeopleSoft Workflow*, Adding Events and Routings, Adding Events.

Truncate

Syntax

```
Truncate(dec, digits)
```

Description

Truncate truncates a decimal number *dec* to a specified precision.

Parameters

Parameter	Description
digits	A Number value that sets the precision of the truncation (that is, the number of digits to leave on the right side of the decimal point).

Returns

Returns a Number value equal to *dec* truncated to a *digits* precision.

Example

The example sets the value of &NUM to 9, 9.99, -9, then 0.

```
&NUM = Truncate(9.9999, 0);
&NUM = Truncate(9.9999, 2);
&NUM = Truncate(-9.9999, 0);
&NUM = Truncate(0.001, 0);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Int , page 298, Chapter 1, PeopleCode Built-in Functions, Mod , page 327, Chapter 1, PeopleCode Built-in Functions, Round , page 381.

Try

Syntax

```
try
Protected StatementList
catch QualifiedID &Id
StatementList
end-try
```

Description

Use the **Try** statement as part of a Try-Catch statement to trap exceptions thrown either by the system or created using the CreateException function.

Parameters

Parameter	Description
Protected StatementList	Specify the statements that are protected by the Try-Catch statement.
catch *QualifiedID &Id*	Specify the **Catch** statement at the end of the list of statements you want to protect.
QualifiedID	Specify what class of exception you are catching, that is, Exception or the name of a class extending it.
&Id	Specify a variable to be set with the caught exception.
StatementList	Specify the steps to be taken once the exception is caught.

Returns

None.

Example

```
try
   &res = 15.3 / 7 * 22.1;
catch Exception &c1
   MessageBox(0, "", 0, 0, "Caught exception: " | &c1.ToString());
end-try;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Throw , page 457, Chapter 1, PeopleCode Built-in Functions, CreateException , page 97.
Chapter 11, Exception Class , page 835

UnCheckMenuItem

Syntax

UnCheckMenuItem(**BARNAME.***menubar_name*, **ITEMNAME.***menuitem_name*)

Description

UnCheckMenuItem removes a check mark from the specified menu item.

Note. This function has been deprecated.

Unencode

Syntax

Unencode(*URLString*)

Description

The **Unencode** function unencodes *URLString*, converting all character codes of the form %xx where xx is a hex number, to the character represented by that number.

Parameters

Parameter	Description
URLString	Specify the string you want unencoded. This parameter takes a string value.

Returns

An unencoded URL string.

Example

For the following example, the Target Content's URL is:

```
http://mrossi101899/PSPortal.jrun/?ICType=Panel&Menu=MANAGE_COMMITMENT_⇒
CONTROL&Markte=GBL&PanelGroupName=KK_NOTIFY_GRP
```

The QueryString parameters are those values of the Target Content's URL that are specified by a &xxx. For example, if you wanted to know the value in the Target Content's URL for the parameter "Menu", then the following PeopleCode would return "MANAGE_COMMITMENT_CONTROL":

```
&MENU = Unencode(%Request.GetParameter("Menu"));
```

This method works for any querystring in the Target Content's URL.

If the link is constructed in a PeopleSoft Internet Architecture page, and the value of a link field, you should not call EncodeURL to encode the entire URL, as the PIA architecture does this for you. You must still unencode the parameter value when you retrieve it, however.

See Also

Chapter 1, PeopleCode Built-in Functions, EncodeURL , page 157, Chapter 1, PeopleCode Built-in Functions, EncodeURLForQueryString , page 158.

Ungray

Syntax

Ungray(*scrollpath*, *target_row*, [*recordname*.]*fieldname*)

where *scrollpath* is:
[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD**.*target_recname*

To prevent ambiguous references, you can also use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

Ungray makes a gray (non-editable) page field editable, if the field was grayed with a call to the **Gray** function.

Note. This function remains for backward compatibility only. Use the **Enabled** Field property instead.

If the page field is made display-only in the Page Field Properties dialog, then **Ungray** has no effect.

The **Gray**, **Ungray**, **Hide**, and **Unhide** functions usually appear in RowInit programs that set up the initial display of data, and FieldChange programs that change field display based on changes the end-user makes to a field.

Generally, you want to put this function on the same scroll level as the field that is being changed in RowInit (which executes on every row) or FieldChange (which executes on the current row). This simplifies the function's syntax to:

```
Ungray(fieldname)
```

A typical use of the more complex syntax is when looping through rows on a scroll on a lower level than the program.

Note. This function shouldn't be used in any event prior to RowInit.

See Also

Chapter 1, PeopleCode Built-in Functions, Gray , page 282, Chapter 1, PeopleCode Built-in Functions, Hide , page 285, Chapter 1, PeopleCode Built-in Functions, Unhide , page 486, Chapter 12, Field Class , Enabled , page 855, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
Scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number of the target row. If this parameter is omitted, the function assumes the row on which the PeopleCode program is executing.
[*recordname* **.**]*fieldname*	The name of the field to ungray. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the call to **Ungray** is in a record definition other than *recordname*.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

The following example checks to see if a person's emergency contact is the same as their home address and phone, then grays or ungrays the fields accordingly. In a typical case, this program would be in the FieldChange event.

```
If SAME_ADDRESS_EMPL = "Y" Then
   Gray(STREET1);
   Gray(STREET2);
   Gray(CITY);
   Gray(STATE);
   Gray(ZIP);
   Gray(COUNTRY);
   Gray(HOME_PHONE);
   STREET1 = PERSONAL_DATA.STREET1;
   STREET2 = PERSONAL_DATA.STREET2;
```

```
      CITY = PERSONAL_DATA.CITY;
      STATE = PERSONAL_DATA.STATE;
      ZIP = PERSONAL_DATA.ZIP;
      COUNTRY = PERSONAL_DATA.COUNTRY;
      HOME_PHONE = PERSONAL_DATA.HOME_PHONE;
   Else
      Ungray(STREET1);
      Ungray(STREET2);
      Ungray(CITY);
      Ungray(STATE);
      Ungray(ZIP);
      Ungray(COUNTRY);
      Ungray(HOME_PHONE);
   End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Gray , page 282, Chapter 1, PeopleCode Built-in Functions, Hide , page 285, Chapter 1, PeopleCode Built-in Functions, Unhide , page 486.

Unhide

Syntax

Unhide(*scrollpath*, *target_row*, [*recordname*.]*fieldname*)

where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]] **RECORD**.*target_recname*

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

Unhide makes a field visible that was previously hidden with **Hide**. If the field was hidden by setting its Invisible property in the Page Field Properties dialog box, then **Unhide** has no effect.

Note. This function remains for backward compatibility only. Use the **Visible** Field property instead.

Generally, you want to put this function on the same scroll level as the field that is being changed in RowInit (which executes on every row) or FieldChange (which executes on the current row). This simplifies the function's syntax to:

```
unhide(fieldname)
```

A typical use of the more complex syntax is when looping through rows on a scroll on a lower level than the program.

Note. This function shouldn't be used in any event prior to RowInit.

See Also

Chapter 1, PeopleCode Built-in Functions, Hide , page 285, Chapter 12, Field Class , Visible, page 870, *PeopleCode Developer's Guide,* Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number of the target row. If this parameter is omitted, the function assumes the row on which the PeopleCode program is executing.
[*recordname* **.**]*fieldname*	The name of the field to unhide. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the call to **Unhide** is in a record definition other than *recordname*.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

The following example sets security for displaying a person's password:

```
If (&DISPLAY) Then
   Unhide(EMPLOYEE.PASSWORD);
Else
   Hide(EMPLOYEE.PASSWORD);
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Gray , page 282, Chapter 1, PeopleCode Built-in Functions, Hide , page 285, Chapter 1, PeopleCode Built-in Functions, Ungray , page 484.

UnhideRow

Syntax

UnhideRow(*scrollpath, target_row*)

Where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,
] **RECORD.***target_recname*

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

UnhideRow programmatically unhides a row that has been hidden by **HideRow**. It unhides the specified row and any dependent rows at a lower scroll level.

Note. This function remains for backward compatibility only. Use the **Visible** Row property instead.

UnhideRow works by putting the row that you unhide to the last non-hidden row in the list. When **UnhideRow** is used in a loop, you have to process rows from low to high to achieve the correct results.

Note. UnhideRow cannot be executed from the same scroll level where the insertion takes place, or from a lower scroll level. Place the PeopleCode in a higher scroll level record.

See Also

Chapter 26, Row Class , Visible, page 1477, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	An integer specifying which row in the scroll to unhide.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

See Chapter 1, PeopleCode Built-in Functions, For , page 192 and Chapter 1, PeopleCode Built-in Functions, Hide , page 285.

```
AE_ROW_COUNT = ActiveRowCount(RECORD.AE_STMT_TBL);
 for &ROW = ActiveRowCount(RECORD.AE_STMT_TBL) to 1 step - 1
    UnhideRow(RECORD.AE_STMT_TBL, &ROW);
    UpdateValue(RECORD.AE_STMT_TBL, &ROW, AE_ROW_NUM, &ROW);
 end-for;
```

See Also

Chapter 1, PeopleCode Built-in Functions, HideRow , page 287.

UnhideScroll

Syntax

UnhideScroll(*Scrollpath*)

Where *scrollpath* is:

[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,] **RECORD**.*target_recname*

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

UnhideScroll programmatically unhides a scroll area that has been hidden with **HideScroll**. It unhides the specified scroll and any associated scrolls at a lower level.

Note. This function remains for backward compatibility only. Use the **ShowAllRows** Rowset method instead.

See Also

Chapter 1, PeopleCode Built-in Functions, HideScroll , page 289, Chapter 27, Rowset Class , ShowAllRows, page 1500, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
scrollpath	A construction that specifies a scroll level in the component buffer.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

This example clears the contents of a level-one hidden scroll, then unhides it:

```
ScrollFlush(RECORD.ORDER_INQ_INV);
 UnhideScroll(RECORD.ORDER_INQ_INV);
```

The following example hides or unhides a level-three scroll:

```
If APPR_QTY_SW = "N" Then
      HideScroll(RECORD.APPR_RULE_LN, CurrentRowNumber(1), RECORD.APPR_RULE_DETL,⇒
 CurrentRowNumber(2), RECORD.APPR_RULE_QTY);
   Else
      UnhideScroll(RECORD.APPR_RULE_LN, CurrentRowNumber(1), ⇒
RECORD.APPR_RULE_DETL, CurrentRowNumber(2), RECORD.APPR_RULE_QTY);
   End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, HideScroll , page 289, Chapter 1, PeopleCode Built-in Functions, RowScrollSelect , page 384, Chapter 1, PeopleCode Built-in Functions, RowScrollSelectNew , page 386, Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389, Chapter 1, PeopleCode Built-in Functions, ScrollSelectNew , page 392, Chapter 1, PeopleCode Built-in Functions, SortScroll , page 440.

UniformSeriesPV

Syntax

UniformSeriesPV(*int_rate*,*n_per*)

Description

UniformSeriesPV calculates the present value of a single monetary unit after a uniform series of payments at a specified interest rate.

Parameters

Parameter	Description
int_rate	A Number specifying the interest rate on the basis of which to calculate the return value.
n_per	A Number specifying the number of payments in the uniform series.

Returns

Returns a Number equal to the value of a single unit after *n_per* payments at an interest rate of *int_rate*.

Example

The example sets &NUM to 3.790786769408448256:

```
&NUM = UniformSeriesPV(10,5);
```

See Also

Chapter 1, PeopleCode Built-in Functions, SinglePaymentPV , page 439.

UpdateSysVersion

Syntax

UpdateSysVersion()

Description

UpdateSysVersion coordinates system changes and changes to system objects maintained by pages, such as messages and Set Tables. This function is not normally used in standard applications and should only used in PeopleSoft-provided extensions of PeopleTools.

Returns

Returns the updated system version Number.

Example

The following example could be used to maintain the version number on MESSAGE_SET_TBL, which controls the refreshing of swap files for the message entries:

```
VERSION = UpdateSysVersion();
```

UpdateValue

Syntax

UpdateValue(*scrollpath*, [*recordname*.]*fieldname*, *target_row*, *value*)

where *scrollpath* is:
[**RECORD.***level1_recname*, *level1_row*, [**RECORD.***level2_recname*, *level2_row*,]]

To prevent ambiguous references, you can use **SCROLL.** *scrollname*, where *scrollname* is the same as the scroll level's primary record name.

Description

UpdateValue updates the value of a specified field with the *value* provided. The value must be of a data type compatible with the *field*.

Note. This function remains for backward compatibility only. Use the **Value** Field property instead.

See Also

Chapter 12, Field Class , Value, page 869, *PeopleCode Developer's Guide*, Data Buffer Access, *PeopleCode Developer's Guide*, Referencing Data in the Component Buffer, Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Parameters

Parameter	Description
Scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	An integer specifying the row of the field to update.
[*recordname* **.**]*fieldname*	The name of the field that you want to update. The field can be on scroll level one, two, or three of the active page. The *recordname* prefix is required if the call to **UpdateValue** is in a record definition other than *recordname*.
Value	The new value to put into the target field.

Returns

None.

Example

This example updates values in the level-one scroll:

```
For &I = 1 To &ROW_CNT
   UpdateValue(RECORD.ASGN_CMP_EFFDT, &I, ITEM_SELECTED, "Y");
End-For;
```

The next example loops through rows in the level-two scroll:

```
For &I = 1 To &CURRENT_L2
   UpdateValue(RECORD.ASGN_CMP_EFFDT, &CURRENT_L1, RECORED.SOME_L2_RECORD, &I, ⇒
TO_CUR, &HOME_CUR);
End-For;
```

See Also

Chapter 1, PeopleCode Built-in Functions, FetchValue , page 181, Chapter 1, PeopleCode Built-in Functions, PriorValue , page 349.

UpdateXmlDoc

Syntax

```
UpdateXmlDoc(&XmlDoc, PubID, PubNode, ChannelName, VersionName [,
Message Name [, SubNode]])
```

Description

Use the **UpdateXmlDoc** function to update a message in the message queue with the specified message version.

If *VersionName* isn't specified, the default message version is used. This method is commonly used in the OnPublishTransform and OnSubscribeTransform PeopleCode events.

Note. This function can't be called from subscription PeopleCode.

Parameters

Parameter	Description
&XmlDoc	Specify an already instantiated XmlDoc object.
PubID	Specify the PubID as a string.
PubNode	Specify the PubNode as a sting.
ChannelName	Specify the Channel name as a string.
VersionName	Specify the version name as a string.
MessageName	Specify the message name as a string. This is only used for Pub and Sub contracts.
SubNode	Specify the sub node as a string. This is only used for Pub contracts.

Returns

A Boolean value: True if function completed successfully, False otherwise.

See Also

Chapter 34, XmlDoc Class , page 1757.

Upper

Syntax

```
Upper(str)
```

Description

Upper converts a text string to all uppercase. This function can be used to perform a case-insensitive string comparison.

Parameters

Parameter	Description
str	A String to convert to uppercase.

Returns

Returns a String value equal to *str* converted to all uppercase.

Example

The following example converts the contents of two string variables to uppercase before determining if they are equal to simulate a case-insensitive comparison:

```
If Upper(&STR1) = Upper(&STR2) Then
   /* do something */
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Lower , page 316, Chapter 1, PeopleCode Built-in Functions, Proper , page 352.

Value

Syntax

Value(*str*)

Description

Value converts a string representing a number to the number.

Parameters

Parameter	Description
str	A String value representing a number.

Returns

Returns the Number value represented by *str*.

Example

The example sets &VAL1 to 5.25 and &VAL2 to 12500:

```
&VAL1 = Value("5.25");
&VAL2 = Value("12,500");
```

See Also

Chapter 1, PeopleCode Built-in Functions, String , page 451.

ViewAttachment

Syntax

ViewAttachment(*URLSource, DirAndFileName, UserFile*)

where *URLSource* can have *one* of the following forms:
URL.*URLname*

OR a string URL, such as
ftp://worker:password@ftp.ps.com/

Description

The **ViewAttachment** function enables an end-user to view a file on a system. The file is actually downloaded into the directory indicated by the **TMP** environment variable.

The URL used to view an attachment can contain non-ASCII character. The viewing file is temporary, and is store on web server's public location. The viewing file may have a different name than the name the end user just typed in.

Note. .exe and .bat files are launched instead of viewed.

Security Considerations

The *FileType* parameter used with AddAttachment is only a suggestion. Specifying it does not enforce specific file types. If you do not want specific file types (such as .exe or .bat) you must write PeopleCode to discern the name and file type of the attached file and delete it before viewing it.

When the end user views attachments using the ViewAttachment function, some browsers treat documents as HTML *regardless of file extension*, and thus execute embedded java script. You may want to write a PeopleCode program to only allow specific file extensions to be viewed.

Restrictions on Use in PeopleCode Events

ViewAttachment is a "think-time" function, which means that it shouldn't be used in any of the following PeopleCode events:

- SavePreChange.
- SavePostChange.
- Workflow.
- RowSelect.
- Any PeopleCode event that fires as a result of a ScrollSelect (or one of its relatives) function calls, or a Select (or one of its relatives) Rowset class method.

See *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Think-Time Functions.

File Name Considerations

When the file is transferred using AddAttachment or PutAttachment, the following characters are replaced with an underscore:

- space

- semicolon
- plus sign
- percent sign
- ampersand
- apostrophe
- exclamation point
- @ sign
- pound sign
- dollar sign

Parameters

Parameter	Description
URLSource	A reference to a URL. This can be either a URL name, in the form URL.*URLName*, or a string.
DirAndFileName	A directory and filename. This is appended to the *URLSource* to make up the full URL of the file.

Note. The *URLSource* requires "/" slashes. Because the *DirAndFileName* parameter is appended to the URL, it also requires only "/" slashes. "\" are NOT supported in anyway for either the *URLSource* the *DirAndFileName* parameter.

Parameter	Description
UserFile	The name of the file on the source system.

Returns

An integer value. You can check for either the integer or the constant:

Numeric Value	Constant Value	Description
0	%Attachment_Success	File was attached successfully.
1	%Attachment_Failed	File was not successfully attached.
2	%Attachment_Cancelled	File attachment didn't complete because the operation was canceled by the user.
3	%Attachment_FileTransferFailed	File transfer didn't succeed.
5	%Attachment_NoDiskSpaceWebServ	No disk space on the web server.
7	%Attachment_DestSystNotFound	Cannot locate destination system for ftp.
8	%Attachment_DestSystFailedLogin	Unable to login to destination system for ftp.
9	%Attachment_FileNotFound	Cannot locate file.
10	%Attachment_DeleteFailed	Cannot delete file.

Example

```
&retcode = ViewAttachment(URL.MYFTP, ATTACHSYSFILENAME, ATTACHUSERFILE);

If (&retcode = %Attachment_Failed) Then
   MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed");
End-If;
If (&retcode = %Attachment_Cancelled) Then
   MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment cancelled");
End-If;
If (&retcode = %Attachment_FileTransferFailed) Then
   MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: File⇒
 Transfer did not succeed");
End-If;

/* following error message only in PeopleSoft Internet Architecture */

If (&retcode = %Attachment_NoDiskSpaceAppServ) Then
   MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: No disk⇒
 space on the app server");
End-If;

/* following error message only in PeopleSoft Internet Architecture */

If (&retcode = %Attachment_NoDiskSpaceWebServ) Then
   MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: No disk⇒
 space on the web server");
End-If;

If (&retcode = %Attachment_FileExceedsMaxSize) Then
   MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: File⇒
 exceeds the max size");
End-If;
If (&retcode = %Attachment_DestSystNotFound) Then
   MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: Cannot⇒
 locate destination system for ftp");
End-If;
If (&retcode = %Attachment_DestSysFailedLogin) Then
   MessageBox(0, "File Attachment Status", 0, 0, "ViewAttachment failed: Unable to⇒
 login into destination system for ftp");
End-If;
```

In the following example, using the file object, a file is written on the application server. Because the absolute path isn't specified, the file is written to PS_SERVDIR/files directory.

The function PutAttachment takes the file from PS_SERVDIR/files directory and puts it on the ftp server specified in URL.GRADEBOOK.

The ViewAttachment function downloads the file into the user's machine and opens it for viewing.

```
/* Data buffer */
Component Rowset &Level0, &Students_Hdr, &Student_Grade;
```

```
Component Row &Level0_Row;

/* standalone rowset - populate from LAM_CLASS_ACTV */
Component Rowset &Class_Activity;
Component number &Class_Act_No;

Local File &Export_File;
Local string &Export_Line;

DoSaveNow();

&File_Name = %OperatorId | "_" | String(CLASS_SCTN_SCTY.STRM) | "_" | String(CLASS_⇒
SCTN_SCTY.CLASS_NBR) | ".txt";

&Export_File = GetFile(&File_Name, "W");

If &Export_File.IsOpen Then
   &Export_Line = "Last,First,Emplid,";
   For &i = 1 To &Class_Activity.ActiveRowCount
      &Export_Line = &Export_Line | &Class_Activity(&i).LAM_CLASS_⇒
ACTV.DESCRSHORT.Value | ",";
   End-For;
   &Export_File.WriteLine(&Export_Line);

   For &i = 1 To &Students_Hdr.ActiveRowCount
      &Export_Line = &Students_Hdr(&i).GetRecord(Record.PERSONAL_DTA_VW).GetField⇒
(Field.NAME).Value | ",";
      &Export_Line = &Export_Line | &Students_Hdr(&i).STDNT_GRADE_HDR.EMPLID.Value⇒
 | ",";
      &Student_Grade = &Level0_Row.GetRowset(Scroll.STDNT_GRADE_HDR)(&i).GetRowset⇒
(Scroll.STDNT_GRADE_DTL);
      For &j = 1 To &Student_Grade.ActiveRowCount
         &Export_Line = &Export_Line | String(&Student_Grade(&j).STDNT_GRADE_⇒
DTL.STUDENT_GRADE.Value) | ",";
      End-For;
      &Export_File.WriteLine(&Export_Line);
   End-For;
   &Export_File.Close();
End-If;

&Put_File_Name = "files/" | &File_Name;
&ret = PutAttachment(URL.GRADEBOOK, &File_Name, &Put_File_Name, "PS_SERVDIR");
&ret = ViewAttachment(GetURL(URL.GRADEBOOK), &File_Name, &File_Name);
```

See Also

Chapter 1, PeopleCode Built-in Functions, AddAttachment , page 31, Chapter 1, PeopleCode Built-in Functions, DeleteAttachment , page 135, Chapter 1, PeopleCode Built-in Functions, GetAttachment , page 234, Chapter 1, PeopleCode Built-in Functions, PutAttachment , page 353, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Understanding File Attachment Architecture.

ViewContentURL

Syntax

ViewContentURL(*URL_str* | URL.*URL_ID*)

Description

Use the **ViewContentURL** function to launch a new browser window and navigates to the location specified by *URL_str* or URL.*URL_ID*.

The content specified by the URL is *not* wrapped by the portal template. Use this function when you want to connect to third-party content. If you want to wrap the content in the portal template, use the **ViewURL** function.

This is a deferred execution command: the browser is launched after any executing PeopleCode has run to completion. This function automatically launches a new browser window.

Note. The **ViewContentURL** function does not work if being run on a Window 95 operating system and Internet Explorer version 3.02 or greater has not been installed.

Parameters

Parameter	Description
URL_str | **URL.** *URL_ID*	Specify the location to where you want to navigate. You can specify *either* a URL string, *or*, a URL saved in the URL table, by specifying the keyword URL followed by a dot and the URL Identifier.

Returns

None.

Example

```
If &MyPage Then
   ViewURL(URL.MYPAGE);
Else
   ViewContentURL("http://www.PeopleSoft.com");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetURL , page 279, Chapter 1, PeopleCode Built-in Functions, ViewURL , page 498.
PeopleSoft Server Tools Administration, PeopleTools Utilities, URL Maintenance

ViewURL

Syntax

ViewURL(*URL_str* | URL.*URL_ID* [, *ReplaceCurrentPage*])

Description

Use the **ViewURL** function to launch the default browser and navigate to the location specified by *URL_str* or URL.*URL_ID*. This is a deferred execution command: the browser is launched after any executing PeopleCode has run to completion. You can also specify whether the new page launches a new browser, or replaces the current page in the browser.

The content specified by the URL is automatically wrapped by the portal template. If you do *not* want to wrap the content in the portal template, use the **ViewContentURL** function.

Note. The **ViewURL** function will not work if being run on a Window 95 operating system and Internet Explorer version 3.02 or greater has not been installed. Portal applications should use the **RedirectURL** Response class method instead of ViewURL.

See Chapter 15, Internet Script Classes (iScript) , RedirectURL, page 953.

Restrictions on Use With Windows Client

The *ReplaceCurrentPage* parameter is not available for Windows Client.

Parameters

Parameter	Description
URL_str | **URL.** *URL_ID*	Specify the location to where you want to navigate. You can specify *either* a URL string, *or*, a URL saved in the URL table, by specifying the keyword URL followed by a dot and the URL Identifier.
ReplaceCurrentPage	Specify whether the new page should replace the current page in the browser, or launch a new browser. This parameter takes a Boolean value: True to replace existing page, False to launch a new browser. The default value is False.

Returns

None.

Example

```
If &MyPage Then
   ViewURL(URL.MYPAGE);
Else
   ViewContentURL("http://www.PeopleSoft.com");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetURL , page 279, Chapter 1, PeopleCode Built-in Functions, ViewContentURL , page 498.
PeopleSoft Server Tools Administration, PeopleTools Utilities, URL Maintenance

Warning

Syntax

Warning *str*

Description

You typically use the **Warning** function in FieldEdit or SaveEdit PeopleCode to display a message alerting the end-user about a potentially incorrect data entry or change. It differs from **Error** in that it does not prevent the end-user from taking an action, and it does not stop processing in the PeopleCode program where it occurs.

Warning is also used in RowDelete and RowSelect PeopleCode, where its behavior is specialized. See the following sections Warnings in RowDelete and Warnings in RowSelect.

The text of the warning message (the *str* parameter), should always be stored in the Message Catalog and retrieved using the **MsgGet** or **MsgGetText** function. This makes it easier to translate the text, and it also enables you to include more detailed Explain text about the warning.

Warnings in FieldEdit and SaveEdit

The primary use of **Warning** is in FieldEdit and SaveEdit PeopleCode:

- In FieldEdit, **Warning** displays a message and highlights the relevant field.
- In SaveEdit, **Warning** displays a message, but does not highlight any field. You can move the cursor to a specific field using the SetCursorPos function.

See Chapter 1, PeopleCode Built-in Functions, SetCursorPos , page 404.

Warnings in RowDelete

When the end-user attempts to delete a row of data, the system first prompts for confirmation. If the end-user confirms, the RowDelete event fires. A **Warning** in the RowDelete event displays a warning message with OK and Cancel buttons. If the end-user clicks OK, the row is deleted. If the end-user clicks Cancel, the row is not deleted.

Warnings in RowSelect

The behavior of **Warning** in RowSelect is totally anomalous and maintained for backward compatibility only. Use it to filter rows being added to a page scroll after the rows have been selected and brought down to the client. **Warning** causes the Component Processor to skip the current row (so that it is not added to the page scroll), then continue processing. No message is displayed.

Note. Do not use **Warning** in this fashion. Use **DiscardRow** for replacement instead.

See Chapter 1, PeopleCode Built-in Functions, DiscardRow , page 146.

Warnings in Other Events

Do not use the **Warning** function in any of the remaining events, which include:

- FieldDefault
- FieldFormula
- RowInit
- FieldChange
- RowInsert
- SavePreChange

- SavePostChange

The end-user has no control over processing that occurs in these events. If the Component Processor encounters a condition that would warrant a warning message in these events, the end-user can't fix it. The Component Processor therefore requires the user to cancel the component to avoid unpredictable results, which results in loss of any changes that the end-user has made.

Parameters

Parameter	Description
str	A string containing the text of the warning message. This string should always be stored in the Message Catalog and retrieved using the **MsgGet** or **MsgGetText** function. This makes translation easier and also enables you to provide detailed Explain text about the warning.

Returns

None.

Example

The following example shows a warning that alerts an end-user to a possible error, but allows the end-user to accept the change:

```
if All(RETURN_DT, BEGIN_DT) and
      8 * (RETURN_DT - BEGIN_DT) < (DURATION_DAYS * 8 + DURATION_HOURS) then
   warning MsgGet(1000, 1, "Duration of absence exceeds standard hours for number⇒
 of days absent.");
end-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Error , page 165, Chapter 1, PeopleCode Built-in Functions, MsgGet , page 328, Chapter 1, PeopleCode Built-in Functions, MsgGetText , page 330, Chapter 1, PeopleCode Built-in Functions, WinMessage , page 505.

Weekday

Syntax

Weekday(*dt*)

Description

Weekday calculates the day of the week based on a date value.

Parameters

Parameter	Description
dt	A Date value. **Weekday** determines the day of the week that *dt* falls on.

Returns

Returns a Number value representing the day of the week. 1 is Sunday, 7 is Saturday.

Example

If &Date_HIRED equals October 30, 1996, a Monday, then the following statement sets &DAY_HIRED to 2:

```
&DAY_HIRED = Weekday(&Date_HIRED);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date , page 117, Chapter 1, PeopleCode Built-in Functions, Date3 , page 118, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Day , page 127, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129, Chapter 1, PeopleCode Built-in Functions, Month , page 327, Chapter 1, PeopleCode Built-in Functions, Year , page 509.

While

Syntax

```
While
 logical_expression
 statement_list
End-while
```

Description

The While loop causes the statements of the *statement_list* to be repeated until *logical_expression* is false. Statements of any kind are allowed in the loop, including other loops. A **Break** statement inside the loop causes execution to continue with whatever follows the end of the loop. If the **Break** is in a nested loop, the **Break** does not apply to the outside loop.

Example

The following example counts from 0 to 10:

```
&COUNTER = 1;
while &COUNTER <= 10
   WinMessage(MsgGet(21000, 1, "Count is %1", &COUNTER));
   &COUNTER = &COUNTER + 1;
end-while;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Repeat , page 372.

WinEscape

Syntax

```
WinEscape( )
```

Description

WinEscape simulates the behavior of the ESC key to cancel a page. This causes any changes to the page subsequent to the last successful save to be lost, and returns the Component Processor to the reset state (the state prior to starting a page or entering a search key).

Restrictions on Use With Internet Client Applications

This function isn't supported by Internet Client.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Example

The following example checks the user input to see if the action requested is valid and cancels the page if the action is invalid:

```
If Substring(%Page, 1, 8) = Substring(PAGE.JOB_DATA1, 1, 8) and
      %Mode = "A" Then
   If All(NAME) Then
      If %Page < PAGE.JOB_DATAX Then
         WinMessage(MsgGet(1000, 81, "The selected Employee ID exists.  It cannot⇒
 be added."));
         WinEscape( );
      End-if;
   Else
      If %Page = PAGE.JOB_DATAX Then
         WinMessage(MsgGet(1000, 82, "The selected Employee ID does not exist. ⇒
 The concurrent Job cannot be added."));
         WinEscape( );
      End-if;
   End-if;
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, EndModal , page 161, Chapter 1, PeopleCode Built-in Functions, Error , page 165, Chapter 1, PeopleCode Built-in Functions, Warning , page 499, Chapter 1, PeopleCode Built-in Functions, DoCancel , page 147.

WinExec

Syntax

WinExec(*command_line*, *window_option* [, *synch_exec*])

Description

WinExec executes a Windows executable file (.COM, .EXE, .BAT, or .PIF) specified in the *command_line* string. It is suitable for executing short batch-processing jobs that will complete execution and terminate, returning control to PeopleTools.

The function can make either a synchronous or asynchronous call. Synchronous execution acts as a modal or “think-time” function, suspending the PeopleSoft application until the called executable completes. This is appropriate if you want to force the end-user (or the PeopleCode program) to wait for the function to complete its work before continuing processing. Asynchronous processing, which is the default, launches the executable and immediately returns control to the calling PeopleSoft application.

Restrictions on Use in PeopleCode Events

When **WinExec** is used to execute a program synchronously (that is, if its *synch_exec* parameter is set to True) it behaves as a think-time function, which means that it can’t be used in any of the following PeopleCode events:

- SavePreChange.
- SavePostChange.
- Workflow.
- RowSelect.
- Any PeopleCode event that fires as a result of a ScrollSelect (or one of its relatives) function calls, or a Select (or one of its relatives) Rowset class method.

See *PeopleCode Developer’s Guide*, Using Methods and Built-In Functions, Think-Time Functions.

Restrictions on Use in Three-Tier Mode and PeopleSoft Internet Architecture

Because it is specific to Windows, **WinExec** is a client-only function, which limits its use in three-tier mode and in PeopleSoft Internet Architecture applications.

- In three-tier mode **WinExec** can be used only in processing groups set to run on the client. If the **WinExec** function is called in a processing group running on the application server, a runtime error occurs.
- In PeopleSoft Internet Architecture applications, where all PeopleCode runs on the application server, **WinExec** cannot be used at all.

In PeopleCode programs that run on the application server, you can use the **Exec** function.

See Chapter 1, PeopleCode Built-in Functions, Exec , page 171.

Parameters

Parameter	Description
window_option	Specifies the style of the newly created window. You can specify either a numeric value or a constant:

Numeric Value	Constant Value	Description
1	%WinExec_Normal	Normal window with focus
2	%WinExec_Minimized	Minimized window with focus
3	%WinExec_Maximized	Maximized window with focus
4	%WinExec_NormalNoFocus	Normal window without focus
7	%WinExec_MaximizedNoFocus	Maximized window without focus

Parameter	Description
command_line	The name of a .COM, .EXE, .BAT, or .PIF file. After the filename, additional command switches or parameters can be specified.
synch_exec	An optional Boolean value specifying whether to execute synchronously (True) or asynchronously (False). The default is asynchronous execution.

Returns

WinExec returns a Number indicating whether the function completed successfully:

Error	Description
0	Completed successfully
non-zero	Error

Example

This example executes, synchronously, a program called control.exe:

```
If INTERNATIONAL_CHG = "Y" Then
   &TMP = WinExec("control.exe", 7, false);
End-if;
```

The following code sample is identical to the previous one, but it uses a constant instead of a numeric value for the window option:

```
If INTERNATIONAL_CHG = "Y" Then
   &TMP = WinExec("control.exe", %WinExec_MaximizedNoFocus, false);
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Declare Function , page 130, Chapter 1, PeopleCode Built-in Functions, Exec , page 171, Chapter 1, PeopleCode Built-in Functions, RemoteCall , page 364.

WinMessage

Syntax

WinMessage(*message* [, *style*] [, *title*])

Description

Note. The **WinMessage** function is supported for compatibility with previous releases of PeopleTools. New applications should use **MessageBox** instead.

See Chapter 1, PeopleCode Built-in Functions, MessageBox , page 319.

The **WinMessage** function displays a message in a message box.

Use the **WinMessage** for simple informational display, where the end-user reads the message, then clicks an OK button to dismiss the message box. **WinMessage** can also be used for branching based on end-user choice, in which case the message box contains two or more buttons (such as OK and Cancel or Yes, No, and Cancel). The value returned by the function tells you which button the end-user clicked, and your code can branch based on that value.

If **WinMessage** displays more than one button, it causes processing to stop while it waits for user response. This makes it a "user think-time" function, restricting its use in certain PeopleCode events.

The contents of the message displayed by **WinMessage** can be passed to the function as a string, but unless you are using the function for testing purposes you should always retrieve the message from the Message Catalog using the **MsgGet** or **MsgGetText** function. This has the advantage of making the messages much easier to localize and maintain.

Restrictions on Use in PeopleCode Events

The *style* parameter is optional in **WinMessage**. If *style* is omitted **WinMessage** displays OK and Cancel buttons, which causes the function to behave as a think-time function. To avoid unnecessary restrictions, you should always pass an appropriate value in the **WinMessage** *style* parameter.

If the *style* parameter specifies a single button (that is, the OK button), the function can be called in any PeopleCode event.

If the *style* parameter specifies more than one button, or if the *style* parameter is omitted, **WinMessage** returns a value based on user response and interrupts processing until the user has clicked one of the buttons. This makes it a "user think-time" function, subject to the same restrictions as other think-time functions which means that it cannot be used in any of the following PeopleCode events:

- SavePreChange.
- Workflow.
- RowSelect.
- SavePostChange.
- Any PeopleCode event that fires as a result of a ScrollSelect (or one of its relatives) function calls, or a Select (or one of its relatives) Rowset class method.

See *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Think-Time Functions.

Restrictions on Use With a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Message Box Icons

In the PeopleSoft Internet Architecture, you can't change the icon of a message box. You can change the number and type of buttons, as well as the default button, but the message always displays with the warning icon (a triangle with an exclamation mark in it.)

Parameters

Parameter	Description
Message	Text displayed in message box. Normally you want to use the **MsgGet** or **MsgGetText** function to retrieve the message from the Message Catalog.
Title	Title of message box.
Style	Either a numerical value or a constant specifying the contents and behavior of the dialog box. This parameter is calculated by cumulatively adding either a value or a constant from each of the following categories:

Category	Value	Constant	Meaning
Buttons	0	%MsgStyle_OK	The message box contains one pushbutton: OK.
	1	%MsgStyle_OKCancel	The message box contains two pushbuttons: OK and Cancel.
	2	%MsgStyle_ AbortRetryIgnore	The message box contains three pushbuttons: Abort, Retry, and Ignore.
	3	%MsgStyle_ YesNoCancel	The message box contains three pushbuttons: Yes, No, and Cancel.
	4	%MsgStyle_YesNo	The message box contains two push buttons: Yes and No.
	5	%MsgStyle_RetryCancel	The message box contains two push buttons: Retry and Cancel.

Note. In Windows Client, if the *style* parameter is omitted, the message window displays an OK and Cancel button with an "i" (Information) icon. The following values for *style* can only be used in Windows Client. They have no affect in PeopleSoft Internet Architecture.

Category	Value	Constant	Meaning
Default Button	0	%MsgDefault_First	The first button is the default.

Category	Value	Constant	Meaning
	256	%MsgDefault_Second	The second button is the default.
	512	%MsgDefault_Third	The third button is the default.
Icon	0	%MsgIcon_None	None
	16	%MsgIcon_Error	A stop-sign icon appears in the message box.
	32	%MsgIcon_Query	A question-mark icon appears in the message box.
	48	%MsgIcon_Warning	An exclamation-point icon appears in the message box.
	64	%MsgIcon_Info	An icon consisting of a lowercase letter "I" in a circle appears in the message box.

Returns

If the *style* parameter is provided, **WinMessage** optionally returns a Number value. If the *style* parameter is omitted, **WinMessage** optionally returns a Boolean value: True if the OK button was clicked, otherwise it returns False.

The return value is zero if there is not enough memory to create the message box.

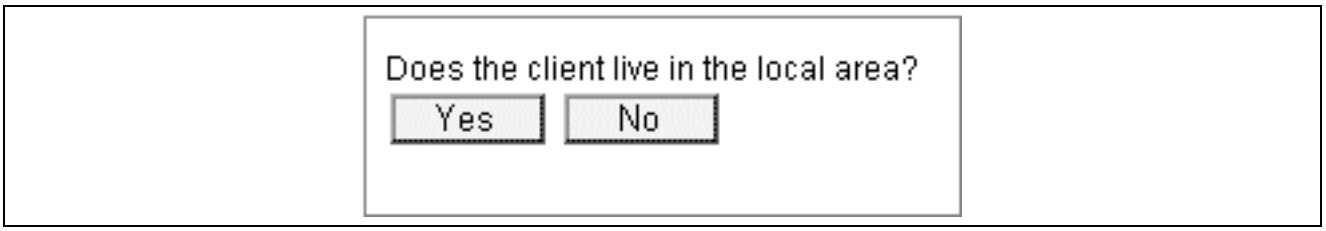
If the *style* parameter is provided, **WinMessage** returns one of the following Number values:

Value	Constant	Meaning
-1	%MsgResult_Warning	Warning was generated.
1	%MsgResult_OK	OK button was selected.
2	%MsgResult_Cancel	Cancel button was selected.
3	%MsgResult_Abort	Abort button was selected.

Value	Constant	Meaning
4	%MsgResult_Retry	Retry button was selected.
5	%MsgResult_Ignore	Ignore button was selected.
6	%MsgResult_Yes	Yes button was selected.
7	%MsgResult_No	No button was selected.

Example

The following example displays a message dialog box with Yes and No buttons. The message is taken from the Message Catalog. The message displayed looks like this:



Message with Yes/No buttons

When the end-user clicks the Yes or No button, a result is passed back which the example tests and uses to control branching.

```
/* Displays Yes/No buttons in message box.  */
&RESULT = WinMessage(MsgGetText(30000, 1, "Message not found."), 4, "Test⇒
 Application");
if &RESULT = %MsgResult_Yes then
   /* Yes button was pressed -- do Yes button stuff */
else
   /* No button was pressed -- do No button stuff */
end-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, Encrypt , page 159, Chapter 1, PeopleCode Built-in Functions, MessageBox , page 319, Chapter 1, PeopleCode Built-in Functions, MsgGet , page 328, Chapter 1, PeopleCode Built-in Functions, MsgGetText , page 330, Chapter 1, PeopleCode Built-in Functions, MsgGetExplainText , page 329.

Year

Syntax

Year(*dt*)

Description

Year derives the year component of a Date value.

Parameters

Parameter	Description
dt	A Date value from which to derive the year component.

Returns

Returns a Number value between 1900 and 2078 equal to the year component in *dt*.

Example

The example sets &GRAD_YEAR to 1976:

```
&GRAD_DATE = DateValue("10/04/76");
&GRAD_YEAR = Year(&GRAD_DATE);
```

See Also

Chapter 1, PeopleCode Built-in Functions, Date , page 117, Chapter 1, PeopleCode Built-in Functions, Date3 , page 118, Chapter 1, PeopleCode Built-in Functions, DateValue , page 126, Chapter 1, PeopleCode Built-in Functions, Day , page 127, Chapter 1, PeopleCode Built-in Functions, Days360 , page 128, Chapter 1, PeopleCode Built-in Functions, Days365 , page 129, Chapter 1, PeopleCode Built-in Functions, Month , page 327, Chapter 1, PeopleCode Built-in Functions, Weekday , page 501.

CHAPTER 2

Meta-SQL

This chapter includes the following topics:

- Date Considerations.
- Meta-SQL Placement Considerations.
- Meta-SQL Reference.

See Also

Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442

Chapter 1, PeopleCode Built-in Functions, ScrollSelect , page 389

Chapter 25, Record Class , page 1441

Chapter 27, Rowset Class , page 1479

Chapter 31, SQL Class , page 1613

Understanding Meta-SQL

Meta-SQL expand to platform-specific SQL substrings, or cause another function to be called, or substitute a value. They are used in functions that pass SQL strings, such as the following:

- SQLExec.
- the scroll buffer functions (ScrollSelect and its relatives).
- in Application Designer to construct dynamic views.
- with some Rowset Class methods (Select, SelectNew, Fill, and so on.).
- with SQL Class.
- in Application Engine.
- with some Record Class methods (Insert, Update, and so on.).
- with COBOL.

Common Terminology Used With Meta-SQL

Construct A direct substitution of a value which helps to build or modify the SQL statement.

	Examples include %Bind, %InsertSelect, %List.
Function	Something that performs an action on its own, or causes another function to be called.
	Examples include %ClearCursor, %Execute, %ExectueEdits.
Meta-variable	Something that allows substitution of text within SQL statements.
	Examples incldue %AsOfDate, %Comma, %JobInstance.

Date Considerations

You can avoid confusion when using meta-SQL such as %Datein and %Dateout if you keep in mind that the "in" functions are used in the WHERE subclause of a SQL query and "out" functions are used in the SELECT (main) clause of the query. For example:

```
select emplid, %dateout(effdt) from ps_car_alloc  a where car_id = '" |⇒
 &REGISTRATION_NO | "' and plan_type = '" | &PLAN_TYPE | "' and a.effdt = ⇒
(select max (b.effdt) from ps_car_alloc b where a.emplid=b.emplid and b.effdt <=⇒
 %currentdatein) and start_dt <= %currentdatein and (end_dt is null or end_dt >=⇒
 %currentdatein)";
```

Use of Date, Datetime, or Time Wrappers with an Application Engine Program

It has always been the expectation in PeopleCode that you should use the date or time wrappers (%Datein, %TimeOut, and so on) when selecting date or time columns into memory. The reason for this is simple: different database platforms use different internal formats for these data types. Those different formats range from 1900-01-01 to 01-JAN-1900. DateTime (timestamp) formats are an even bigger deal.

In PeopleCode (SQLExecs and the like), the developer must use both an "Out" wrapper when selecting a date/time value into memory, as well as an "In" wrapper when referencing the value as a bind variable.

In an Application Engine program, things are different. When you populate a date/time state field in a %Select, you still have to use an "Out" wrapper to get the value into the standard format. But when you reference this state field in a %Bind, Application Engine automatically provides the "In" wrapper around the substituted literal or bind marker (the latter if reuse is in effect).

Actually, Application Engine does a bit more: if you use the %Bind(date) in the select list of another %Select statement, to load the value into another date field, Application Engine doesn't provide any wrapper (since you are selecting a value that is already in the standard format, you don't need to use any wrapper.)

Date and Time Out Wrappers for Static and Dynamic Views

Dynamic views containing date, time, or datetime fields *must* be wrapped with the appropriate Meta-SQL. PeopleTools uses the SQL directly from the view definition (View Text) and doesn't generate anything, so no Meta-SQL wrapping is done.

For a normal view (that is, a static view) PeopleTools builds the SELECT statement and subsequently wraps Date, Time and DateTime columns with the appropriate Meta-SQL. Normal Views should *not* contain Meta-SQL (otherwise the system adds the Meta-SQL to a field that has already been passed through Meta-SQL and the statement fails).

Using {DateTimein-prefix} in SQR

In SQR, if you are using {DateTimein-prefix} , and so on, you need to do the following:

- For string or let statements when using dynamic SQL, you need to use the following:

```
{DYN-Date***in/out-prefix/suffix}
```

- For SQL statements, you need to use the regular SQL, as follows:

```
{Date*** in/out-prefix/suffix}
```

Meta-SQL Placement Considerations

Not all meta-SQL can be used by all programs. Some meta-SQL can be used in Application Engine. Other meta-SQL can only be used as part of a SQL statement in a record view. The following table contains a list of all the available meta-SQL. An X in the appropriate column means that you can use that meta-SQL in that type of program.

If a meta-SQL construct, function or meta-variable is supported in PeopleCode, it is supported in all types of PeopleCode programs, that is, in Application Engine PeopleCode programs (actions), Application Message PeopleCode programs, and so on.

Note. "Used in PeopleCode" does *not* mean that you can use meta-SQL like a function. It means that you can use the meta-SQL in **SQLExec**, the **Select** method, the **Fill** method, and so on.

Note. You should only place meta-SQL in a Dynamic View, not in a SQL view.

Meta-SQL that is only available for Application Engine are fully described in the PeopleSoft Application Engine documentation.

Name	Used in PeopleCode	Used in Application Engine	Used in COBOL	Used in Dynamic View
%Abs	X	X		X
%BINARYSORT	X	X	X	X
%Bind		X		

Name	Used in PeopleCode	Used in Application Engine	Used in COBOL	Used in Dynamic View
%ClearCursor		X		
%COALESCE	X	X		X
%Concat	X	X		X
%CurrentDateIn	X	X	X	X
%CurrentDateOut	X	X	X	X
%CurrentDateTimeIn	X	X	X	X
%CurrentDateTimeOut	X	X	X	X
%CurrentTimeIn	X	X	X	X
%CurrentTimeOut	X	X	X	X
%DateAdd	X	X	X	X
%DateDiff	X	X	X	X
%DateIn	X	X	X	X
%DateNull	X	X		X
%DateOut	X	X	X	X
%DatePart	X	X		X
%DateTimeDiff	X	X	X	X
%DateTimeIn	X	X	X	X
%DateTimeNull	X	X		X
%DateTimeOut	X	X	X	X

Name	Used in PeopleCode	Used in Application Engine	Used in COBOL	Used in Dynamic View
%DecDiv	X	X	X	X
%DecMult	X	X	X	X
%Delete	X			
%DTTM	X	X		X
%EffDtCheck	X	X		
%Execute		X		
%ExecuteEdits		X		
%FirstRows	X	X		
%Insert	X			
%InsertSelect	X	X		X
%InsertValues	X			X
%Join	X	X		X
%KeyEqual	X			X
%KeyEqualNoEffDt	X			X
%Like	X	X		X
%LikeExact	X	X		X
%List		X		
%ListBind		X		
%ListEqual		X		

Name	Used in PeopleCode	Used in Application Engine	Used in COBOL	Used in Dynamic View
%Next and %Previous		X		
%NoUppercase	X	X		X
%NumToChar	X	X		X
%OldKeyEqual	X			X
%OPRCLAUSE				X
%Round	X	X	X	X
%RoundCurrency		X		
%Select		X		
%SelectAll	X			
%SelectByKey	X			
%SelectByKeyEffDt	X			
%SelectDistinct	X			
%SelectInit		X		
%SQL	X	X		X
%Substring	X	X	X	X
%SUBREC				X
%Table	X	X		X
%Test	X	X		
%TextIn	X	X		X

Name	Used in PeopleCode	Used in Application Engine	Used in COBOL	Used in Dynamic View
%TimeAdd	X	X		
%TimeIn	X	X	X	X
%TimeNull	X	X		X
%TimeOut	X	X	X	X
%TimePart	X	X		X
%TrimSubstr	X	X	X	X
%Truncate	X	X	X	X
%TruncateTable	X	X	X	X
%Update	X			
%UpdatePairs	X			X
%UpdateStats		X		
%Upper	X	X		X

See Also

PeopleSoft Application Engine, Using Meta-SQL and PeopleCode

Meta-SQL Reference

Throughout this section, we use typographical conventions to distinguish between different elements of the PeopleCode language, such as bold to indicate function names, italics for arguments, and so on.

Note. The parameter *recname* refers to a *record* name, *not a table name*. If you specify a table name (for example, PS_ST_OPTION_PARMS) you'll receive a SQL error. You should use the record name (for example, ST_OPTION_PARMS) instead. Also, don't use quotation marks around a record name.

See Also

Chapter 1, PeopleCode Built-in Functions, PeopleCode Typographical Conventions, page 1.

%Abs

Syntax

%Abs(*x*)

Description

%Abs returns a decimal value equal to the absolute value of a number *x*.

Note. This meta-SQL is not implemented for COBOL.

Example

```
SELECT INVENTORY_CODE FROM INVENTORY_TABLE WHERE %ABS(NEW_AMOUNT - OLD_AMOUNT) >⇒
 SOME_ALLOWED_VALUE
```

%BINARYSORT

Syntax

%BINARYSORT(*Recname*)

Description

Any in-memory sorting performed using COBOL language functions will be performed as a binary sort in the current character set used for COBOL processing, and may not necessarily match the sort order returned by the database in response to an ORDER BY clause. Should you require the database to return data sorted using a binary sort of its encoding rather than the default linguistically-correct sort, you must use the %BINARYSORT Meta-SQL function around each column in the WHERE or ORDER BY clause where binary ordering is imporant.

However, for OS/390 implementations, you must keep in mind that this binary sorting is only equivalent when the COBOL program is run OS/390 server. For example, the binary sort produced in COBOL differs from the binary sort produced by the database, as the database is encoded in EBCDIC and the client is in an ASCII-based encoding. Therefore, %BINARYSORT should only be used in COBOL programs that are not run using RemoteCall where the OS/390 platform is not supported as a RemoteCall server.

When running against non-OS/390 systems, %BINARYSORT can be used in both RemoteCall and non-RemoteCall programs.

Note. Using %BINARYSORT in your WHERE and ORDER BY clauses negates the use of any indexes, as most databases can’t use indexes for functional comparisons. (For example, WHERE %BINARYSORT(column) > ’X’). Use this syntax only when sorting equivalence of SQL statement results and COBOL memory order is absolutely required.

Parameters

Parameter	Description
Recname	Specify the record name to be used with the sorting.

Example

```
SELECT RECNAME FROM PSRECDEFN  WHERE %BINARYSORT(RECNAME) < %BINARYSORT('xxx')

SELECT RECNAME FROM PSRECDEFN  ORDER BY %BINARYSORT(RECNAME)
```

See Also

Chapter 1, PeopleCode Built-in Functions, RemoteCall , page 364.
PeopleTools Global Technology, COBOL Globalization

%COALESCE

Syntax

%COALESCE(*expr1*, *expr2*, ...)

Description

Use the **%COALESCE** function to return the first non-null argument provided to the function.

Note. This meta-SQL is not implemented for COBOL.

Parameters

Parameter	Description
expr1. . .exprn	Specify the expressions that you want checked.

Example

The following example uses the PRODUCT_INFO table to organize a "clearence sale" of products. It give a 10% discount to all products with a list price. If there is no list price, the sale price is the minimum price. If there is no minimum price, the sale price is 10.

```
SELECT product_id, list_price, min_price, %COLESCE(0.9*list_price, min_price, 10)⇒
 "Sale"
from PRODUCT_INFO
where SUPPLIER_ID = 6009;
```

%Concat

Syntax

string1 **%Concat** *string2*

Description

At runtime, the **%Concat** meta-variable is replaced by the string concatenation operator appropriate for the RDBMS being executed on. For example, on DB2, the %Concat meta-variable is replaced with CONCAT, while on Sybase it's replaced with a "+".

This meta-variable is supported with the same limitations as the native concatenation operator on the RDBMS the meta-SQL is being executed on. For example, some platforms allow you to concatenate a string with a numeric value; others flag this as an error. PeopleTools makes no attempt to check or convert the data types of either of the operands.

Note. Concat is not available in COBOL, but the **DYN-STMT-CONCAT** field can be strung into dynamic COBOL strings to resolve into a platform specific concatenation operator.

Example

```
SELECT 'A' %Concat 'B' FROM PS_INSTALLATION. . .

SELECT LAST_NAME %Concat ',' %Concat FIRST_NAME FROM PS_EMPLOYEE
```

%CurrentDateIn

Description

%CurrentDateIn expands to a platform-specific SQL substring representing the current date in the WHERE clause of a SQL SELECT or UPDATE statement, or when the current date is passed in an INSERT statement.

%CurrentDateOut

Description

%CurrentDateOut expands to platform-specific SQL for the current date in the SELECT clause of an SQL query.

%CurrentDateTimeIn

Description

%CurrentDateTimeIn expands to a platform-specific SQL substring representing the current datetime in the WHERE clause of a SQL SELECT or UPDATE statement, or when the current date time is passed in an INSERT statement.

%CurrentDateTimeOut

Description

%CurrentDateTimeOut expands to platform-specific SQL for the current datetime in the SELECT clause of an SQL query.

%CurrentTimeIn

Description

%CurrentTimeIn expands to a platform-specific SQL substring representing the current time in the WHERE clause of a SQL SELECT or UPDATE statement, opr when the current time is passed in an INSERT statement.

%CurrentTimeOut

Description

%CurrentTimeOut expands to platform-specific SQL for the current time in the SELECT clause of an SQL query.

%DateAdd

Syntax

%DateAdd(*date_from*, *add_days*)

Description

%DateAdd returns a date by adding *add_days* to *date_from*. *add_days* can be negative.

Example

```
SQLExec("Select %dateadd(%datein('2002-02-02') , 12) from PS_INSTALLATION_TR",⇒
 &add);
WinMessage(&add);
```

%DateDiff

Syntax

%DateDiff(*date_from*, *date_to*)

Description

%DateDiff returns an integer representing the difference between two dates in number of days.

Example

```
%DateDiff(%DateIn('1997-01-01'), %DateIn("1966-06-30'))

%DateDiff( date1_column, date2_column)

%DateDiff ( %DateAdd(date1_column, 30), date2_column)
```

The following usage is **illegal** (always use %Datein for inputting date literals).

```
%DateDiff('1997-01-01', '1996-06-30') (should use %DateIn for inputting date⇒
 literals)
```

%DateIn

Syntax

%DateIn(*dt*)

Description

%DateIn expands into platform-specific SQL syntax for the date. %DateIn should be used whenever a date literal or Date bind variable is used in a comparison in the WHERE clause of a SELECT or UPDATE statement, or when a Date value is passed in an INSERT statement.

Restrictions using COBOL

You can only use string literals when using this construct in COBOL. You can *not* use it with bind parameters in COBOL. For example, the following works in COBOL:

```
UPDATE PS_PERSONAL_DATA SET LASTUPDT = %DATEIN('2002-12-11')
```

Whereas the following SQL fails:

```
UPDATE PS_PERSONAL_DATA SET LASTUPDT = %DATEIN(:1)
```

Parameters

Parameter	Description
dt	Specify either a Date value or a date literal in YYYY-MM-DD format

%DateNull

Syntax

%DateNull

Description

Use this meta-SQL when you want to specify a Null for a date field. Only use this meta-SQL in INSERT or UPDATE clauses. Do *not* use this meta-SQL in a WHERE clause.

Note. This meta-SQL is not implemented for COBOL.

This meta-SQL resolves into a database specific SQL substring.

Database	Resolves into
Informix	empty string (’’)
DB2	NULLIF(CURRENT DATE, CURRENT DATE)
All others	NULL

Parameters

None.

%DateOut

Syntax

%DateOut(*dt*)

Description

%DateOut expands to a platform-specific SQL substring representing *dt* in the SELECT clause of an SQL query.

Parameters

Parameter	Description
dt	Specify *dt* as a date column.

%DatePart

Syntax

%DatePart(*DTTM_Column*)

Description

%DatePart returns the date portion of the specified datetime column.

Note. This meta-SQL is not implemented for COBOL.

Considerations using %DatePart

Use %DateOut meta-SQL when fetching values:

```
%DateOut(%DatePart(DTTM_COLUMN)) from some_table
```

If a literal is used as the parameter to %DatePart, it must be wrapped in %DateTimeIn:

```
insert into some_table values(%DatePart(%DateTimeIn('2001-01-01-12.34.56.789012')))
```

Parameters

Parameter	Description
DTTM_Column	Specify the datetime column you want just the date part returned

%DateTimeDiff

Syntax

%DateTimeDiff(*datetime_from*, *datetime_to*)

Description

%DateTimeDiff returns a time value, representing the difference between two datetimes in minutes.

Example

The following example returns the difference in hours between the current datetime and the requested date time:

```
%DateTimeDiff(%CurrentDateIn, RQSTDTTM) < " | RECORD.FIELDNAME * 60;
```

The following example returns the difference in minutes:

```
%DateTimeDiff(%CurrentDateIn, RQSTDTTM) < " | RECORD.FIELDNAME;
```

%DateTimeIn

Syntax

%DateTimeIn(*dtt*)

Description

%DateTimeIn(*dtt*) expands to platform-specific SQL for a DateTime value in the WHERE clause of a SQL SELECT or UPDATE statement, or when a datetime value is passed in an INSERT statement.

Restrictions using COBOL

You can only use string literals when using this construct in COBOL. You can *not* use it with bind parameters in COBOL. For example, the following works in COBOL:

```
UPDATE PS_PERSONAL_DATA SET LASTUPDTTM = %DATETIMEIN('2002-12-11-11.59.00.000000')
```

Whereas the following SQL fails:

```
UPDATE PS_PERSONAL_DATA SET LASTUPDTTM = %DATETIMEIN(:1)
```

Parameters

Parameter	Description
dtt	Specify either a Datetime bind variable or a String literal in the form YYYY-MM-DD-hh.mm.ss.ssssss

%DateTimeNull

Syntax

%DateTimeNull

Description

Use this meta-SQL when you want to specify a NULL for a datetime field. Only use this meta-SQL in INSERT or UPDATE clauses. Do *not* use this meta-SQL in a WHERE clause.

Note. This meta-SQL is not implemented for COBOL.

This meta-SQL resolves into a database specific SQL substring.

Database	Resolves into
Informix	empty string (’’)
DB2	NULLIF(CURRENT TIMESTAMP, CURRENT TIMESTAMP)
All others	NULL

Parameters

None.

Example

```
%InsertSelect(LEDGER_KK_WK2,LEDGER_KK_WRK, CURRENCY_CD = %Bind(TO_CURRENCY)⇒
 ,POSTED_TOTAL_AMT = SUM(POSTED_BASE_AMT),POSTED_TRAN_AMT = 0,POSTED_BASE_AMT =⇒
 0,BASE_CURRENCY = %Bind(TO_CURRENCY),PROCESS_INSTANCE = %Bind(PROCESS_⇒
INSTANCE),DTTM_STAMP_SEC = %DateTimeNull)

FROM PS_LEDGER_KK_WRK

WHERE PROCESS_INST_STG = %Bind(PROCESS_INSTANCE)

AND CURRENCY_CD <> %Bind(TO_CURRENCY)

GROUP BY PROCESS_INST_STG, BUSINESS_UNIT,LEDGER, ACCOUNT, %List(FIELD_LIST, ⇒
CFCC1_AK_SBR) ,STATISTICS_CODE, FISCAL_YEAR,ACCOUNTING_PERIOD
```

%DateTimeOut

Syntax

%DateTimeOut(*datetime_col*)

Description

%DateTimeOut expands to a platform-specific SQL substring representing *datetime_col* in the SELECT clause of an SQL query.

Parameters

Parameter	Description
datetime_col	Specify a Datetime column.

%DecDiv

Syntax

%DecDiv(*a,b*)

Description

%DecDiv returns a floating number representing the value of a divided by b, where a and b are numeric expressions.

If the result needs to be picked up by a bind variable, it is recommended to pick it up using character type or PIC X(50).

Parameters

Parameter	Description
a, *b*	Numeric expressions.

Example

```
%DecDiv(1000.0, :1)
```

where :1 is a bind variable in SQLExec PeopleCode.

%DecMult

Syntax

%DecMult(*a*,*b*)

Description

%DecMult returns a floating number representing *a* multiplied by *b*, where a and b are numeric expressions.

If the result needs to be picked up by a bind variable, it is recommended to pick it up using character type or PIC X(50).

Parameters

Parameter	Description
a, *b*	Numeric expressions.

Example

```
%DecMult(12.3, 34.67)

%DecMult(c1 + c2, c3)
```

where c1, c2 and c3 are fields of NUMBER datatype.

%DTTM

Syntax

%DTTM(*date*, *time*)

Description

%DTTM combines the database date in the *date* value with the database time in the *time* value and returns a database timestamp value.

Note. This meta-SQL is not implemented for COBOL.

Example

```
INSERT INTO TABLE1 (TIMESTAMP) SELECT %DTTM(DATE,TIME) FROM TABLE2
```

%EffDtCheck

Syntax

%EffDtCheck(*recordname* [*correlation_id1*], *correlation_id2, as_of_date*)

Description

The **%EffDtCheck** construct expands into an effective date sub-query suitable for a WHERE clause. The value for *as_of_date* will automatically be wrapped in %Datein() unless *as_of_date* is already wrapped in %Datein() or refers to other database columns.

Note. This meta-SQL is not implemented for COBOL.

%EffDtCheck only works with effective dates. It does *not* take effective sequence numbers (EFFSEQ) into account. It also doesn't do any effectie status (EFF_STATUS) checking.

Parameters

Parameter	Description
recordname	Specify the record name to be used as the record in the effective date checking. This can be a bind variable, a record object, or a record name in the form *recname*. You *cannot* specify a RECORD.*recname*, a record name in quotes, or a table name. **Note.** If you specify a bind variable, it should refer to a record object, *not* a string variable.
correlation_id1	Specify the letter used inside the effective dating subselect. This is an optional parameter. If it isn't specified, *recordname* is used.
correlation_id2	Specify the letter already assigned to the main record in the FROM clause of the SQL statement.
as_of_date	Specify the date to be used in the effective date. This can be a bind variable, a variable or a hard-coded date. The value for *as_of_date* will automatically be wrapped in %Datein().

Example

The following is a generic code sample:

```
SELECT. . .
   FROM. . .
      WHERE %EffDtCheck(recordname, correlation_id, as_of_date)
```

resolves into the following:

```
SELECT . . .
   FROM. . .
   WHERE correlation_id.EFFDT = (SELECT MAX(EFFDT) FROM recordname
      WHERE recordname.KEYFIELD1 = correlation_id.KEYFIELD1
      AND recordname.KEYFIELD2 = correlation_id.KEYFIELD2
      AND. . .
      AND recordname.EFFDT <= %DATEIN(as_of_date))
```

In the following example, &Date has the value of 01/02/1998. And the example &Rec object has an EFFDT key field. The following code sample:

```
SQLExec("SELECT FNUM FROM PS_REC A WHERE %EffDtCheck(:1, A, :2)", &Rec, &Date);
```

resolves into the following:

```
"Select FNUM from PS_REC A where EFFDT = (select MAX(EFFDT)
from PS_REC
   where PS_REC.FNUM = A.FNUM
   and PS_REC.EFFDT <= %DateIn('1998-01-02') )"
```

The following example uses correlation ids. The following code sample:

```
SELECT A.DEPTID
FROM %Table(DEPT_TBL) A
WHERE
%EffDtCheck(DEPT-TBL B, A, %CurrentDateIn)
AND A.EFF_STATUS = 'A'
```

resolves into the following:

```
SELECT A.DEPTID
FROM %Table(DEPT_TBL) A
WHERE
A.EFFDT =
(SELECT MAX(B.EFFDT)
FROM DEPT_TBL B
WHERE
A.SETID = B.SETID
AND A.DEPTID = B.DEPTID
AND B.EFFDT <=%CurrentDateIn)
AND A.EFF_STATUS = 'A'
```

%FirstRows

Syntax

%FirstRows(*n*)

Description

The **%FirstRows** meta-SQL is replaced by the database specific SQL syntax to optimize retrieval of *n* rows. Depending on the database, it will optimize:

- The query path.
- The number of rows returned.
- The number of rows returned per fetch buffer.

Considerations using %FirstRows

Consider the following when using %FirstRows:

- %FirstRows does *not* mean that only the first X rows are returned. It means that the SQL is optimized for the first X rows where the platform supports it. More rows might be returned, depending on the platform.
- It is the application's responsibility to stop fetching when it has had enough rows.
- This meta-variable is *not* implemented for COBOL or Dynamic View SQL.
- This meta-variable is *not* implemented for Sybase. It will do nothing on this platform.
- This meta-variable *must not be used* if there's a chance that the application requires more than *n* rows fetched. The results of fetching more than *n* rows varies by platform. Some return the extra rows, but performance may be sub-optimal. Others return "ROW NOT FOUND".
- This meta-variable must only be between the SELECT that begins the SQL statement and the SELECT LIST. It must *not* be used in subqueries, views, INSERT/SELECTS, and so on. The SELECT list must *not* be *.
- This meta-variable must not be used with DISTINCT, because SELECT TOP 1 DISTINCT will fail on SqlServer.
- This meta-variable is implicitly embedded in all SELECT SqlExecs for all platforms except ORACLE.

Parameters

Parameter	Description
n	Specify the number of rows to be returned.

Example

The following is checking for the existence of a row:

```
&SQL = CreateSQL("select %firstrows(1) 'x' from PS_EXAMPLE where COL1 = :1",⇒
 &temp);
```

The following populates a 10 element array:

```
&SQL = CreateSQL("select %firstrows(10) COL2, COL3 from PS_EXAMPLE_VW where COL1 =⇒
 :1", &temp);
```

%InsertSelect

Syntax

%InsertSelect([DISTINCT,]*insert_recname*, *select_recname* [*correlation_id*][, *select_recname_n* [*correlation_id_n*]] [, *override_field* = *value*]. . .)

Description

The **%InsertSelect** construct generates an INSERT statement with a SELECT for you. It does *not* generate the FROM statement. You must specify all the select records before you specify any override fields.

Note. InsertSelect has a limit of 99 override fields.

The INSERT column list is composed of all the fields in the specified *insert_recname*, with the exception of LONG_CHAR or IMAGE fields.

Note. Because of the way longs (long character and image fields) are handled in the various database platforms for INSERT statements, all longs in the *insert_recname* will be skipped in the generated INSERT statement. This implies that these fields should be defined in such a manner as to allow Nulls.

The corresponding value in the SELECT list is generated based on the following precedence:

1. If the INSERT fieldname appears as an *override_field*, the corresponding *value* will be used in the SELECT list.
2. If the INSERT fieldname matches a fieldname in one of the *select_recname* specified, the corresponding SELECT field will be used in the SELECT list.
3. The search order of the *select_recname* records is the order that they are specified in the %InsertSelect.
4. If the INSERT fieldname has a constant default value defined in Application Designer, that value is used in the SELECT list.
5. A default value appropriate for the data type of the INSERT field is used (blank for character, zero for numerics, Null for date/time/datetime values, and so on.)

The optional *override_field* is used to specify values for a particular field. For each field you specify, the matching logic described above is not performed. Instead, the value you specify after the equal sign is used for that field in the actual SELECT list. Use this technique when you want to let PeopleTools or Application Engine handle most of the fields in the record, but need to specify some of them explicitly. Also, you can use *override_field* to specify aggregate functions like sum( ), max( ), and so on.

Note. This meta-SQL is not implemented for COBOL.

Parameters

Parameter	Description
DISTINCT	Optionally specify if the select statement being generated should contain the word DISTINCT.
insert_recname	Specifies the name of record being inserted into. You must specify a record name, *not* a RECORD.*recname*, a record name in quotes, a bind variable or a table name. **Note.** If the record for *insert_recname* is a temporary table, %InsertSelect automatically substitutes the corresponding table instance (PS_TARGET*nn* instead of PS_TARGET).
select_recname	Specifies the name of record being selected from. You can specify more than one record to be selected from. You must specify a record name, not a RECORD.*recname*, a record name in quotes, or a table name.
correlation_id	Identifies the correlation ID to be used for the *select_recname* records and fields.
override_field	Specify the name of a field on *insert_recname* that you want to supply a value for (instead of using the value supplied from the *select_recname*.)
Value	Specify the value that should be used for the *override_field* instead of the value from *select_recname.*

Example

The following code:

```
%InsertSelect(AE_SECTION_TBL, AE_STEP_TBL S, AE_SECTION_TYPE = ' ')
   FROM PS_AE_STEP_TBL S, PS_AS_STMT_TBL T
WHERE. . .
```

will resolve into the following:

```
INSERT INTO PS_AE_SECTION_TBL (AE_APPLID, AE_SECTION,. . ., AE_SECTION_TYPE)
SELECT S.AE_APPL_ID, S.AE_SECTION, . . . ' '
FROM PS_AE_STEP_TBL S, PS_AS_STMT_TBL T
   WHERE. . .
```

Another example would be the case where you have a temporary table, PS_MY_TEMP, which is based on a join between two of your other tables, PS_MY_TABLE1 and PS_MY_TABLE2.

```
%InsertSelect(MY_TEMP, MY_TABLE1, MY_TABLE2 T2)
   FROM PS_MY_TABLE1 T1, PS_MY_TABLE2 T2
WHERE %Join(COMMON_KEYS, MY_TABLE1 T1, MY_TABLE2 T2) . . .
```

This code will resolve into:

```
INSERT INTO PS_MY_TEMP (FIELD1, FIELD2 . . .)
   SELECT T2.FIELD1, T2.FIELD2, . . .
FROM PS_MY_TABLE1 T1, PS_MYTABLE2 T2
WHERE T1.FIELD1 = T2.FIELD1
AND T1.FIELD2 = T2.FIELD2 . . .
```

The following example assigns a SQL statement to a variable:

```
Function insert_draft_type(&RECNAME)

&SQL = "%InsertSelect(" | &RECNAME | " ," | &RECNAME | ", SETID = :2, DRAFT_TYPE =⇒
 :3) FROM %Table(:1) WHERE " | FIELD.SETID | " = :4 and " | FIELD.DRAFT_TYPE | " =:⇒
5";

SQLExec(&SQL, @("RECORD." | &RECNAME), SETID, DRAFT_TYPE, DRAFT_TYPE_TABLE.SETID,⇒
 DRAFT_TYPE_TBL.DRAFT_TYPE);

End-Function;
```

The following example will create a distinct select.

```
%InsertSelect(DISTINCT, MY_TABLE, TABLE1, TABLE2 T2)
   FROM PS_TABLE1 T1, PS_TABLE2 T2
WHERE %Join(COMMON_KEYS, TABLE1 T1, TABLE2 T2) . . .
```

This code will resolve into:

```
INSERT INTO PS_MYTABLE (FIELD1, FIELD2 . . .)
   SELECT DISTINCT T2.FIELD1, T2.FIELD2, . . .
FROM PS_TABLE1 T1, PS_TABLE2 T2
WHERE T1.FIELD1 = T2.FIELD1
AND T1.FIELD2 = T2.FIELD2 . . .
```

%InsertValues

Syntax

%InsertValues(*recname*)

Description

The **%InsertValues** construct produces a comma separated list of the record's nonnull field values. Input processing is applied to the fields, that is:

- If the field is a date, a time, or a datetime, its value will automatically be wrapped in %Datein(), %TimeIn(), or %DateTimeIn(), respectively.
- If the field is a string, its value will automatically be wrapped in quotes.
- If the field has a null value, it will not be included in the list.

Note. This meta-SQL can only be used in PeopleCode programs, not in Application Engine SQL actions. Also, this meta-SQL is not implemented for COBOL.

Parameters

Parameter	Description
recname	Specify the name of the record to be used for inserting. This can be a bind variable, a record object, or a record name in the form *recname*. You *can't* specify a RECORD.*recname*, a record name in quotes, or a table name.

Example

The following code:

```
SQLExec("Insert into TABLE (%List(NonNull_Fields, :1)) values (%InsertValues(:⇒
1))", &Rec);
```

will be expanded into:

```
"Insert into TABLE (FNUM, FCHAR, FDATE) values (27, 'Y', %datein('1989-11-27'))"
```

%Join

Syntax

%Join({COMMON_KEYS | COMMON_FIELDS}, join_recname
[*correlation_id1*], *to_recname* [*correlation_id2*]
[, *override_field_list*])

where *override_field_list* is an arbitrary-length list of fields to be substituted in the resulting text string, in the form:
field1 [, *field2*]. . .

Description

Use the **%Join** construct to dynamically build a WHERE clause joining one table to another. At runtime, the entire construct is replaced with a character string.

Note. This meta-SQL is not implemented for COBOL. If date key fields are not marked as required in the record definition for either of the referenced tables in the %Join clause, a Null clause check is added to the date field compare. This additional clause can have a significant impact on the execution time for the generated SQL statement.

Parameters

Parameter	Description
COMMON_KEYS	Specifies that all common primary key fields will be used in constructing a WHERE clause. You can select either COMMON_KEYS or COMMON_FIELDS.
COMMON_FIELDS	Specifies that all common fields will be used in constructing a WHERE clause. You can select either COMMON_KEYS or COMMON_FIELDS.
join_recname	Specifies the name of the record to be joined. This can be a bind variable, a record object, or a record name in the form *recname*. You *can't* specify a RECORD.*recname*, a record name in quotes, or a table name.
correlation_id1	Identifies the correlation ID to be used to relate the record specified by *join_recname* and its fields.
to_recname	Specifies the name of the record to be joined to. This can be a bind variable, a record object, or a record name in the form *recname*. You *can't* specify a RECORD.*recname*, a record name in quotes, or a table name.
correlation_id2	Identifies the correlation ID to be used to relate the record specified by *to_recname* and its fields.
override_field_list	Specifies a list of any fields that you do not want used in the join. For example, if fields A, B, and C were common to two records, and you didn't want to join on C, list C as an *override_field*.

Example

The following code:

```
%Join(COMMON_KEYS, PSAESECTDEFN ABC,  PSAESTEPDEFN XYZ)
```

will result in the following being generated:

```
ABC.AE_APPLID = XYZ.AE_APPLID
AND ABC.AE_SECTION = XYZ.AE_SECTION
AND ABC.DBTYPE = XYZ.DBTYPE
AND ABC.EFFDT = XYZ.EFFDT
```

The following code:

```
%Join(COMMON_FIELDS, PSAEAPPLDEFN ABC,  PSAESECTDEFN XYZ)
```

will result in the following being generated:

```
ABC.AE_APPLID = XYZ.AE_APPLID
AND ABC.DESCR = XYZ.DESCR
```

However, you don't want to join using the DESCR. So, use the *override_field*, as shown in the following code:

```
%Join(COMMON_FIELDS, PSAEAPPLDEFN ABC,  PSAESECTDEFN XYZ, DESCR)
```

to get:

```
ABC.AE_APPLID = XYZ.AE_APPLID
```

You can also specify a value for a field. Suppose you want to join two tables, but not on the field C3. In addition, you would like to specify a value for C3. Your code could look like the following:

```
%Join(COMMON_FIELDS, MY_TABLE1 A, MY_TABLE2 B, C3) AND C3 = 'XX'
```

%KeyEqual

Syntax

%KeyEqual(*recname* [*correlation_id*])

Description

The **%KeyEqual** construct expands into a conditional phrase suitable for use in a WHERE clause.

The conditional phrase consists of a conjunction (AND) of *[correlation_id.]keyfieldname* = '*keyfieldvalue*' phrases for each key field of the given record.

No autoupdate processing is done, but other input processing is applied to the values. That is:

- If the field is a date, a time, or a datetime, its value will automatically be wrapped in %Datein(), %TimeIn(), or %DateTimeIn(), respectively.
- If a value is a string, its value will automatically be wrapped in quotes.
- If a value is NULL, the "=*value*" part is replaced with "IS NULL".

Note. This meta-SQL can only be used in PeopleCode programs, not in Application Engine PeopleCode actions. Also, this meta-SQL is not implemented for COBOL.

Parameters

Parameter	Description
recname	Specify the name of the record to be used for inserting. This can be a bind variable, a record object, or a record name in the form *recname*. You *can't* specify a RECORD.*recname*, a record name in quotes, or a table name.
correlation_id	Identifies the single letter correlation ID to be used to relate the record specified by *recname* and its fields.

Example

The record &REC has three keys: FNUM, FDATE, FSMART. Therefore, the following code:

```
Local record &REC;

&REC = CreateRecord(RECORD.MYRECORD);
&REC.FNUM.Value = 27;
&REC.FDATE.Value = %Date;
SQLExec("Delete from MYRECORD A where %KeyEqual(:1, A)", &REC);
```

expands to:

```
"Delete from TABLE A
   where A.FNUM = 27
```

```
AND A.FDATE = %Date('1989-11-27')
AND A.FSMART IS NULL"
```

%KeyEqualNoEffDt

Syntax

%KeyEqualNoEffDt(*recname* [*correlation_id*])

Description

The **%KeyEqualNoEffDt** construct expands into a conditional phrase suitable for use in a WHERE clause.

The conditional phrase consists of a conjunction (AND) of *[correlation_id.]keyfieldname = 'keyfieldvalue'* phrases for all key fields of the given record, *except* that it omits any key field named EFFDT.

No autoupdate processing is done, but other input processing is applied to the values. That is:

- If the field is a date, a time, or a datetime, its value will automatically be wrapped in %Datein(), %TimeIn(), or %DateTimeIn(), respectively.
- If a value is a string, its value will automatically be wrapped in quotes.
- If a value is NULL, the "=*value*" part is replaced with "IS NULL."

Note. This meta-SQL can only be used in PeopleCode programs, not in Application Engine PeopleCode actions. Also, this meta-SQL is not implemented for COBOL.

Parameters

Parameter	Description
recname	Specify the name of the record to be used for inserting. This can be a bind variable, a record object, or a record name in the form *recname*. You *can't* specify a RECORD.*recname*, a record name in quotes, or a table name.
correlation_id	Identifies the single letter correlation ID to be used to relate the record specified by *recname* and its fields.

Example

The EMPL_CHECKLIST record has three keys: EMPLID, CHECK_SEQ, and EFFDT. Therefore, the following code:

```
&REC = CreateRecord(EMPL_CHECKLIST);

SQLExec("Delete from TABLE A where %KeyEqualNoEffdt(:1, A)", &REC)
```

expands to:

```
"Delete from TABLE A
   where A.EMPLID = 8001
   AND A.CHECK_SEQ = 00001"
```

%Like

Syntax

%Like("*Literal*")

Description

The **%Like** construct expands to look for literal values. This meta-SQL should be used when looking for like values. A % is appended to *literal*.

Note. This meta-SQL is not implemented for COBOL.

If you're using a bind marker (such as ":1") for the literal argument in a SQLExec, you must wrap the SQL string with **ExpandSqlBinds**. ExpandSqlBinds replaces bind markers with the actual input values.

%Like generates the following:

```
like 'literal%'
```

If the literal value contains '_' or '\%', %Like generates the following:

```
like 'literal%' escape '\'
```

See Chapter 1, PeopleCode Built-in Functions, ExpandSqlBinds , page 178.

Using %Like and Eliminating Blanks

Some platforms require that you use RTRIM to get the correct value. The following characters are wildcards even when preceded with the backslash '\' escape character.

- %
- _

Therefore, on some platforms, the literal must end with a '%' wildcard that isn't preceded by '\'.

```
literal = 'ABC%' --> no need for RTRIM on any platform

literal = 'ABC\%' --> need RTRIM on MSS and DB2.
```

Using %Like and Trailing Blanks

It is important to note that not all executions of LIKE perform the same. When dealing with trailing blanks, some platforms behave as if there is an implicit % at the end of the comparison string, while most don't, as shown in the following example:

In this example, if the selected column contains the string "ABCD " (three trailing blanks), the following statement may or may not return the any rows:

```
select  *  from  t1 Where c like 'ABCD'
```

Therefore, it is always important to explicitly code the % the end of matching strings for those columns where you want to include trailing blanks.

DBMS	Implicit %?
PeopleSoft Standard Usage	Always use %
DB2/400	No
DB2/MVS	No
DB2/Unix	No
Informix	No
Microsoft SQL Server	Yes
Oracle	No
SQLBase	No
Sybase	Yes

LIKE Wild Cards

SQL specifies two wild cards that can be used when specifying pattern matching strings for use with the LIKE predicate. The underscore is used as a substitution for a single character within a string, the percent sign is used to represent any number of character spaces within a string.

DBMS	Single Character	Multiple Characters
PeopleSoft Standard Usage	_	%
DB2/400	_	%
DB2/MVS	_	%
DB2/Unix	_	%
Informix	_	%
Microsoft SQL Server	_	%
Oracle	_	%

DBMS	Single Character	Multiple Characters
SQLBase	_	%
Sybase	_	%

Parameters

Parameter	Description
literal	Specify the value you want to search for.

%LikeExact

Syntax

%LikeExact(*fieldname*, "*Literal*")

Description

The **%LikeExact** meta-SQL expands to look for literal values. This should be used when exact matches are necessary, taking into account wildcards in the literal values.

Note. This meta-SQL is not implemented for COBOL.

%LikeExact generates one of the following:

- If the literal contains no wildcards:

```
fieldname = 'literal'
```

- If the literal ends with the '%' wildcard:

```
fieldname like 'literal' [escape '\']
```

Some platforms require that you use RTRIM to get the correct value. The following characters are wildcards even when preceded with the backslash '\' escape character.

- %
- _

Therefore, on some platforms, the literal must end with a '%' wildcard that isn't preceded by '\'.

```
literal = 'ABC%' --> no need for RTRIM on any platform

literal = 'ABC\%' --> need RTRIM on MSS and DB2.
```

Considerations using Bind Markers

If you're using a bind marker (such as ":1") for the literal argument in a SQLExec, you must wrap the SQL string with ExpandSqlBinds. ExpandSqlBinds replaces bind markers with the actual input values.

The following forms work:

- AE SQL (with or without Reuse enabled)

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID, ⇒
%Bind(AE_APPL_ID, STATIC))
```

  The STATIC modifier is only required if Reuse is enabled, but it can't hurt to use it (if Reuse is not enabled, static is the default anyway).

- PeopleCode

```
AE_TESTAPPL_AET.AE_APPL_ID = "AB\_C";

SQLExec("UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID, ⇒
:AE_TESTAPPL_AET.AE_APPL_ID)");
```

  or

```
SQLExec(ExpandSqlBinds("UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LikeExact⇒
(AE_APPL_ID, :1)", "AB\_C"));
```

This form does *not* work:

```
SQLExec("UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID, ⇒
:1)", "AB\_C");
```

See Also

Chapter 1, PeopleCode Built-in Functions, ExpandSqlBinds , page 178.

Parameters

Parameter	Description
fieldname	Specify a field to be used in the first part of the LIKE comparison.
literal	Specify the value you want to search for.

Example

The following code

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID, 'ABC')
```

resolves into the following:

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE AE_APPL_ID = 'ABC'
```

The following code

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID, 'AB%C')
```

resolves into the following:

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE RTRIM(AE_APPL_ID) LIKE 'AB%C'
```

The following code

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE LIKEEXACT(AE_APPL_ID, 'AB%C%')
```

resolves into the following:

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE AE_APPL_ID LIKE 'AB%C%'
```

The following code

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE %LIKEEXACT(AE_APPL_ID, 'AB%C% ')
```

resolves into the following:

```
UPDATE PS_AE_APPL_TMP SET AE_PRODUCT = 'X' WHERE AE_APPL_ID LIKE 'AB%C% '
```

The following example shows using ExpandSqlBinds:

```
SQLExec(ExpandSqlBinds("SELECT COUNT(*) FROM PS_ITEM WHERE %LIKEEXACT⇒
(BUSINESS_UNIT, :1)", "M04"), %COUNT);
```

%List

Syntax

%List({FIELD_LIST | FIELD_LIST_NOLONGS | KEY_FIELDS | ORDER_BY },
recordname [*correlation_id*])

Description

The **%List** construct expands into a list of field names, delimited by commas. Which fields are included in the expanded list depends on the parameters passed to the construct.

Note. This meta-SQL is not implemented for COBOL, Dynamic View SQL or PeopleCode.

Parameters

Parameter	Description
FIELD_LIST	Use all field names in the given record. You can select only one option from FIELD_LIST, ORDER_BY, FIELD_LIST_NOLONGS or KEY_FIELDS..
KEY_FIELDS	Use all key fields in the given record. You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS, KEYFIELDS or ORDER_BY.
ORDER_BY	Use all the key fields of *recordname*, adding DESC for descending key columns. This parameter is often used when the list being generated is for an ORDER BY clause. You can select only one option from FIELD_LIST, KEY_FIELDS, ORDER_BY, or FIELD_LIST_NOLONGS.
FIELD_LIST_NOLONGS	Use all field names in the given record, *except* any "long" columns (that is, long text or image fields.) You can select only one option from FIELD_LIST, ORDER_BY, KEY_FIELDS, or FIELD_LIST_NOLONGS.
recordname	Identifies either a record or a subrecord that the field names will be drawn from. This can be a bind variable, a record object, or a record name in the form *recname*. You *cannot* specify a RECORD.*recname*, a record name in quotes, or a table name.
correlation_id	Identifies the single letter correlation ID to be used to relate the record specified by *recordname* and its fields.

Considerations Using %List

When using %List in an insert/select or insert/values or %Select statement, you *must* have matching pairs of %List (or %ListBind) in the target and source field lists, using the same list type argument and record name to ensure consistency.

Example

The following is a *good* example of using %List. Note that both the INSERT and SELECT statements use the same %List.

```
INSERT INTO PS_PO_DISTRIB_STG (  %Sql(POCOMMONDISTSTGFLDLSTU)
,  %List(FIELD_LIST, CF16_AN_SBR)
,  MERCHANDISE_AMT
,  MERCH_AMT_BSE
,  QTY_DEMAND
,  QTY_PO
,  QTY_PO_STD
,  QTY_REQ)
SELECT  %Sql(POCOMMONDISTSTGFLDLSTU)
,  %List(FIELD_LIST, CF16_AN_SBR)
,  MERCHANDISE_AMT
,  MERCH_AMT_BSE
,  QTY_DEMAND
,  QTY_PO
,  QTY_PO_STD
,  QTY_REQ
FROM PS_PO_DIST_STG_WRK WRK
WHERE WRK.PROCESS_INSTANCE = %Bind(PROCESS_INSTANCE)
```

The following example shows both a *poor* example of how to use %List, as well as how the example might be rewritten to use %List correctly.

In the following SQL Object, INSERT and SELECT field lists both use %List, but the SELECT field list is only partly dynamic. The rest is hard-coded.

Poor code example:

```
INSERT INTO PS_EN_TRN_CMP_TMP (%List(FIELD_LIST, EN_TRN_CMP_TMP))
SELECT B.EIP_CTL_ID
, %List(SELECT_LIST, EN_BOM_COMPS A)
, E.COPY_DIRECTION
, E.BUSINESS_UNIT_TO
, E.BOM_TRANSFER_STAT
, 'N'
, B.MASS_MAINT_CODE
, 0
 FROM PS_EN_BOM_COMPS A
 , PS_EN_ASSY_TRN_TMP B
 , PS_EN_TRNS_TMP E
WHERE ...
```

Rewritten *Good* code example:

```
INSERT INTO PS_EN_TRN_CMP_TMP (EIP_CTL_ID,
,  %List(FIELD_LIST, EN_BOM_COMPS)
,  COPY_DIRECTION
,  BUSINESS_UNIT_TO
,  BOM_TRANSFER_STAT
,  EN_MMC_UPDATE_FLG
,  MASS_MAINT_CODE
,  EN_MMC_SEQ_FLG01
,  ...
, EN_MMC_SEQ_FLG20)
SELECT B.EIP_CTL_ID
  , %List(FIELD_LIST, EN_BOM_COMPS A)
, E.COPY_DIRECTION
, E.BUSINESS_UNIT_TO
, E.BOM_TRANSFER_STAT
, 'N'
, B.MASS_MAINT_CODE
, 0
, ...
, 0
 FROM PS_EN_BOM_COMPS A
 , PS_EN_ASSY_TRN_TMP B
 , PS_EN_TRNS_TMP E
WHERE ...
```

The following example shows both a *poor* example of how to use %List, as well as how the example might be rewritten to use %List correctly.

In the following SQL Object, only the field list of on the INSERT portion is dynamically generated, but the SELECT is statically coded. If the table STL_NET_TBL is reordered, the INSERT will be incorrect.

Example of *poor* code:

```
INSERT INTO PS_STL_NET_TBL (%List(FIELD_LIST, STL_NET_TBL ) )
SELECT :1
, :2
, :3
, :4
, :5
, :6
, :7
,:8
FROM PS_INSTALLATION
```

Rewritten *Good* code example

```
INSERT INTO PS_STL_NET_TBL (%List(FIELD_LIST, STL_NET_TBL))

VALUES (%List(BIND_LIST, STL_NET_TBL MY_AET))
```

%ListBind

Syntax

```
%ListBind({FIELD_LIST | FIELD_LIST_NOLONGS | KEY_FIELDS}, recordname
[ State_record_alias])
```

Description

The **%ListBind** construct expands a field list as bind references for use in an INSERT/VALUE statement.

Note. This meta-SQL is not implemented for COBOL, Dynamic View SQL or PeopleCode.

Considerations Using %ListBind

When using %List in an insert/select or insert/values or %Select statement, you *must* have matching pairs of %List or %ListBind in the target and source field lists, using the same list type argument and record name to ensure consistency.

Parameters

Parameter	Description
FIELD_LIST	Use all field names in the given record. You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS or KEY_FIELDS.
FIELD_LIST_NOLONGS	Use all field names in the given record, *except* any "long" columns (i.e., long text or image fields.) You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS or KEY_FIELDS.
KEY_FIELDS	Use all key field names in the given record. You can select only one option from FIELD_LIST, FIELD_LIST_NOLONGS or KEY_FIELDS.
recordname	Identifies either a record or a subrecord that the field names will be drawn from. This can be a bind variable, a record object, or a record name in the form *recname*. You *cannot* specify a RECORD.*recname*, a record name in quotes, or a table name.
State_record_alias	Specify the Application Engine State Record buffer that contains the values (this could be different than the record used to derive the field list.) If missing, the default State Record is assumed.

Example

```
INSERT INTO PS_TARGET (FIELD1, FIELD2, %List(FIELD_LIST, CF_SUBREC), FIELDN)⇒
 VALUES (%Bind(MY_AET.FIELD1), %Bind(MY_AET.FIELD2), %ListBind(FIELD_LIST, ⇒
CF_SUBREC MY_AET), %Bind(MY_AET.FIELDN))
```

%ListEqual

Syntax

```
%ListEqual({ALL | KEY }, Recordname [alias], RecordBuffer [, Separator])
```

Description

The **%ListEqual** construct maps each field, possibly to an alias, with a %Bind value, with a separator added before each equality. Each field is mapped as follows:

```
<alias>.X = %Bind(<recbuffer>.X)
```

This construct can be used in the SET clause of an update or in a WHERE clause.

Note. This meta-SQL is not implemented for COBOL, Dynamic View SQL or PeopleCode.

Parameters

Parameter	Description
ALL | **KEY**	Specify the type of fields you want, whether you want all the fields, or just the key fields.
recordname	Identifies either a record or a subrecord that the field names will be drawn from. This can be a bind variable, a record object, or a record name in the form *recname*. You *cannot* specify a RECORD.*recname*, a record name in quotes, or a table name.
alias	Specify an optional alias to prefix each field name.
RecordBuffer	Specify the record buffer for the binds (this could be different than the record used to derive the field list.)
Separator	Specify a separator to be used between each equality. Valid values are AND or OR. The default value of a comma will be used if no separator is specified.

Example

```
UPDATE PS_TEMP

SET %ListEqual(ALL, CF_SUBREC, MY_AET)
WHERE %ListEqual(KEYS, TEMP, MY_AET, AND)
```

%NoUppercase

Syntax

%NoUppercase

Description

When processing a SQL statement, the system automatically casts all fieldnames and possibly record names to uppercase when processing a SQL statement. When processing records from a third party, fields that are lowercase get cast into uppercase, which can create a runtime issue on case sensitive platforms.

To prevent this, use the %NoUppercase meta-SQL statement at the beginning of the SQL statement.

Parameters

None.

Note there are not parameters, as well as no parenthesis, for this meta-SQL.

Returns

None.

Example

```
%NoUppercase
INSERT INTO PS_RM_APP_ENG_LOG (MAP_ID
, RECNAME
, FIELDNAME
, MESSAGE_SET_NBR
, MESSAGE_NBR
, LANGUAGE_CD)
SELECT %Bind(MAP_ID)
, %Bind(RECNAME)
, ' '
,17834
, 1116
, %Bind(LANGUAGE_CD)
FROM PS_INSTALLATION
WHERE EXISTS (
SELECT 'X'
FROM SW_OPPORTUNITY SW_OPPORTUNITY
, SW_PERSON SW_PERSON
, SW_CUSTOMER SW_CUSTOMER
, SW_SALES_TEAM_VW SW_SALES_TEAM_VW
WHERE SW_OPPORTUNITY.SWCUSTOMERID = SW_CUSTOMER.SWCUSTOMERID
AND SW_OPPORTUNITY.SWSALESTEAMID = SW_SALES_TEAM_VW.SWPROVIDERGRPID
AND SW_SALES_TEAM_VW.SWPERSONID = SW_PERSON.SWPERSONID
GROUP BY SW_OPPORTUNITY.SwOpportunityId
HAVING COUNT(*) > 1)
```

%NumToChar

Syntax

%NumToChar(*Number*)

Description

Use the **%NumToChar** construct to transform a numeric value into a character value. Spaces are trimmed from *Number*.

Parameters

Parameter	Description
Number	Specify the number you want converted to a character value. Signed numbers, as well as decimals, are accepted.

%OldKeyEqual

Syntax

%OldKeyEqual(*recname [correlation_id]*)

Description

The **%OldKeyEqual** construct is similar to the **%KeyEqual** construct, except that it uses the "original" values of the record fields, rather than the current values. Since the rules by which values are original and which are current are not very clear, especially for stand-alone record objects, the use of this meta-SQL construct is discouraged. You should use separate records to hold the desired previous values. This will make your code much clearer and more maintainable.

Note. This meta-SQL can only be used in PeopleCode programs, not in Application Engine PeopleCode actions. Also, this meta-SQL is not implemented for COBOL.

See Also

Chapter 2, Meta-SQL , %KeyEqual , page 535.

%OPRCLAUSE

Description

The %OPRCLAUSE metastring is used in the viewtext of dynamic views. In PeopleTools 6 the %OPRCLAUSE metastring expanded in the following manner:

```
SELECT EMPLID, ABSENCE_TYPE, oprid
FROM PS_ABSENCE_HIST
WHERE %OPRCLAUSE

SELECT EMPLID,  ABSENCE_TYPE,  OPRID  FROM PS_ABSENCE_HIST WHERE ( OPRCLASS ='⇒
HRADMIN') AND (EMPLID='8001' AND ABSENCE_TYPE='CNF') ORDER BY EMPLID, ABSENCE_TYPE
```

In PeopleTools 7, to support the new concept of a specific "Row Level Security Class", this metastring now fills in the WHERE clause with the value from PSOPRDEFN.ROWSECCLASS.

%OPRCLAUSE must be either all upper case or all lowercase. Mixed case isn't allowed.

%OPRCLAUSE translates to OprId or OprClass, following the same rules used for security on search dialogs. If OPRID is in the view %OPRCLAUSE expands to OPRID = 'current operator'. If OPCLASS is in the view %OPRCLAUSE expands to OPCLASS = 'current class'.

For example the following:

```
SELECT EMPLID, ABSENCE_TYPE, OPRID FROM PS_ABSENCE_HIST WHERE %OPRCLAUSE AND ⇒
(EMPLID='8001' AND ABSENCE_TYPE='CNF')
```

expands to:

```
SELECT EMPLID,  ABSENCE_TYPE,  OPRID  FROM PS_ABSENCE_HIST WHERE ( OPRID =⇒
 'PTDMO') AND (EMPLID='8001' AND ABSENCE_TYPE='CNF') ORDER BY EMPLID, ABSENCE_TYPE
```

For example, the following:

```
SELECT EMPLID, ABSENCE_TYPE, OPRCLASS FROM PS_ABSENCE_HIST WHERE %OPRCLAUSE AND ⇒
(EMPLID='8001' AND ABSENCE_TYPE='CNF')
```

expands to:

```
SELECT EMPLID,  ABSENCE_TYPE,  OPRID  FROM PS_ABSENCE_HIST WHERE ( OPRCLASS =⇒
 'ALLPANLS') AND (EMPLID='8001' AND ABSENCE_TYPE='CNF') ORDER BY EMPLID, ⇒
ABSENCE_TYPE
```

%Round

Syntax

%Round(*expression*, *factor*)

Description

%Round rounds an expression to a specified scale before or after the decimal point. If *factor* is a literal, it can be rounded to a negative number.

Parameters

Parameter	Description
expression	An arbitrary numeric expression involving numeric constants and database columns.
factor	An integer or bind variable in SQLExec PeopleCode. The range of a factor is from -31 to +31 for literals. Non-literals can only be positive.

Example

```
%Round(10.337, 2) = 10.34

%Round(13.67, 0) = 14

SQLExec("SELECT %Round(field_c1, :1) from RECORD_T", field_c2, &Result);
```

where field_c1 and field_c2 are two fields in the record.

The following cases are illegal, and may cause incorrect results or runtime SQL errors:

```
%Round(10.337, 2 + 1)  (factor can not be an expression)

%Round(field_c1, field_c2) (factor can not be database columns)
```

%SQL

Syntax

%SQL(*SQLid* [, *paramlist*])

where *paramlist* is a list of arguments that are used for dynamic substitutions at runtime, in the form: *arg1* [, *arg2*]. . .

Description

Use the **%SQL** construct for common SQL fragments that you have already defined and want to re-use, substituting additional values dynamically. *SQLid* is the name of a SQL definition created using either Application Designer or the **StoreSQL** function.

You can only nest up to 10 %SQL statements at a time.

Note. This meta-SQL is not implemented for COBOL. A SQL definition is *not* the same as the SQL Object that is instantiated from the SQL Class at runtime. A SQL definition is created either using Application Designer at design time, or using the StoreSQL function. A SQL Object is instantiated at runtime from the SQL Class, and has methods and properties associated with it like any other object.

When a SQL definition specified has more than one version, the database type always takes precedence. That is:

- If one or more versions of a SQL definition are found for the database type of the current database connection, and if any of the versions have an effective date less than or equal to the value returned for %AsOfDate, the most recent version is used.
- If no versions are found for the current database type, or if all of the versions have effective dates greater than the value returned for %AsOfDate, the system looks for an effective version of the SQL definition under the database type "generic". If no version is found, an error occurs.

See *PeopleCode Developer's Guide*, Introducing the SQL Editor.

Application Engine Considerations

Application Engine programs use the current date to compare with the effective date, not the date returned by %AsOfDate.

Special SQL Characters

The following meta-SQL meta-variables can be used as part of the %SQL construct to represent special characters as SQL parameters.

%Comma	Represents a single comma.
%LeftParen	Allows you to pass a left parenthesis, "(", to a %P() variable, without closing the SQL object.
%RightParen	Allows you to pass a right parenthesis, ")", to a %P() variable, without closing the SQL object.
%Space	Represents a space.

Parameters

Parameter	Description
SQLid	Specify the name of an existing SQL definition.
paramlist	An optional list of arguments that are used for dynamic substitutions at runtime. The first argument replaces all occurrences of %P(1) in the referenced SQL definition, the second argument replaces %P(2), and so forth. You can specify up to 99 arguments in paramlist.

Example

In the following example, the SQL definition MY_SQL was created in Application Designer to be the following:

```
%P(1).EFFDT = (SELECT MAX(EFFDT) FROM ...)
```

Therefore the following code (the **bold** portions will be dynamically generated):

```
UPDATE PS_TEMP
SET ...
WHERE ...
AND %SQL(MY_SQL, PS_TEMP)
```

resolves to the following:

```
UPDATE PS_TEMP
SET ...
WHERE ...
AND PS_TEMP.EFFDT = (SELECT MAX(EFFDT) FROM ...)
```

See Also

Chapter 31, SQL Class , page 1613

%Substring

Syntax

%Substring(*source_str*, *start*, *length*)

Description

%Substring expands to a substring of *source_str*.

Parameters

Parameter	Description
source_str	Specify the source string.
start	Specify the substring's beginning position. The first character of *source_str* is position 1.
length	Specify the length of the substring.

%SUBREC

Syntax

%SUBREC(*subrec_name, corel_name*)

Description

%SUBREC is used only in Dynamic View SQL where it expands to the columns of a subrecord: you can't use this statement in SQLExec or any other SQL statement.

Note. SUBREC must be either all upper case or all lowercase. Mixed case isn't allowed.

Parameters

Parameter	Description
subrec_name	Specify the name of the subrecord.
corel_name	Specify the correlation name.

Example

Suppose you have a record definition AAA_VW that is a dynamic view with fields CHR, SUB, and NUM. The field SUB is a subrecord with fields CHR_SUB, NUM_SUB, and IMG_SUB. The viewtext for AAA_VW could be:

```
"select a.chr, %subrec(sub,a), a.num from ps_aaa a"
```

The Create View SQL generated by this viewtext would be:

```
"CREATE VIEW SYSADM.PS_AAA_VW (CHR, CHR_SUB, NUM_SUB, IMG_SUB, NUM) AS SELECT⇒
 A.CHR, A.CHR_SUB, A.NUM_SUB, A.IMG_SUB, A.NUM FROM PS_AAA A"
```

%Table

Syntax

%Table(*recname* [*, instance*])

Description

The **%Table** construct returns the SQL table name for the record specified with *recname*.

For example, %Table(ABSENCE_HIST) will return PS_ABSENCE_HIST.

Note. This meta-SQL is not implemented for COBOL.

If the record is a temporary table and the current process has a temp table instance number assigned, %Table resolves to that instance of the temp table (that is, PS_ABSENCE_HIST*nn*, where *nn* is the instance number).

Or, you can override this value with the *instance* parameter. For example, suppose you know you want the third instance of a temp table, you could specify it with %Table(&MYREC, 3). You can use the SetTempTableInstance function to set the instance of a temp table that will be used with %Table.

This construct can be used to specify temporary tables for running parallel Application Engine processes.

See Also

PeopleSoft Application Engine, Using Meta-SQL and PeopleCode, %Table.

Parameters

Parameter	Description
recname	Identifies the record that the table name will be drawn from. This can be a bind variable, a record object, or a record name in the form *recname*. You *cannot* specify a RECORD.*recname*, a record name in quotes, or a table name.
instance	Specify the instance of the temp table to be used.

Example

The following function deletes records based on two other fields:

```
Function delete_draft_type(&RECNAME)

&SQL = "Delete from %Table(:1) where " | FIELD.SETID | " =
 :2 and " | FIELD.DRAFT_TYPE | " = :3";

SQLExec(&SQL, @("RECORD." | &RECNAME), SETID, DRAFT_TYPE);

End-Function;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetTempTableInstance , page 431.

%Test

Syntax

%Test(*Prefix*, *Test*, *Suffix*)

Description

The **%Test** construct can be used with records that have no key values.

Parameters

Parameter	Description
Prefix	Specify a string that is conditionally added before the expansion of the test string. You can *not* use meta-SQL in this parameter.
Test	Specify a meta-SQL string to be expanded.
Suffix	Specify a string that is conditionally added at the end of the test string. You can use meta-SQL in this parameter.

Returns

If the expansion of *Test* produces only a blank (or empty) string, the entire %Test meta-SQL construct is replaced with nothing. Otherwise the %Test meta-SQL construct is replaced by the prefix, then the expansion of *Test*, then the suffix.

Example

For example, the following meta-SQL generates valid SQL even when the given record has no keys:

```
%SelectAll(:1) %Test(WHERE ,%KeyEqual(:1));
```

%TextIn

Syntax

%TextIn(*BindVariable*)

Description

%TextIn construct, when used with a bind variable, allows the insertion and updating of a text string into a LONG CHARACTER field (column).

Note. This Meta-SQL is not implemented for COBOL. This Meta-SQL is mandatory for any LONG CHARACTER field insertion or update to be compatible on all database platforms. If you don't use this Meta-SQL wrapper, this type of operation fails on Sybase and Informix.

Parameters

Parameter	Description
BindVariable	A bind variable

Example

In the following example, :1 is a bind variable in PeopleCode.

```
&String1 = "This is a test."

SqlExec("INSERT INTO PS_TABLE1 (STMTID, SQLSTMT) VALUES (1, %TextIn(:1))",⇒
 &String1)
```

%TimeAdd

Syntax

%TimeAdd(*datetime*, *add-minutes*)

Description

This construct generates the SQL that will add *add-minutes* (which can be a positive or negative integer literal or expression, provided that the expression resolves to a data type that can be used in datetime arithmetic for the given RDBMS) to the provided *datetime* (which can be a datetime literal or expression).

Note. On some platforms, you can use a *time-value* in place of *datetime*. However, this can give a SQL error on other platforms (for example, Informix) if the result of the %TimeAdd would result in a new date (for example, 11:59PM + 2 minutes). This meta-SQL is not implemented for COBOL.

Parameters

Parameter	Description
time	Specify a time or datetime value that you want to add more time to.
addminutes	Specify the number of minutes you want to add to *time*. This must be a numeric value or an expression that resolves to a numeric value.

Example

```
SELECT %TimeAdd(%CurrentTimeIn, 60) FROM PS_INSTALLATION
```

%TimeIn

Syntax

%TimeIn(*tm*)

Description

%TimeIn expands to platform-specific SQL for a Time value in the WHERE clause of a SQL SELECT or UPDATE statement, or when a time value is passed in an INSERT statement.

Restrictions using COBOL

You can only use string literals when using this construct in COBOL. You can *not* use it with bind parameters in COBOL. For example, the following will work in COBOL:

```
UPDATE PS_PERSONAL_DATA SET LASTUPTM = %TIMEIN('11:59:00:000000')
```

Whereas the following SQL will fail:

```
UPDATE PS_PERSONAL_DATA SET LASTUPTM = %TIMEIN(:1)
```

Parameters

Parameter	Description
tm	Specify a Time bind variables or a String literal in the form: hh.mm.ss.ssssss

%TimeNull

Syntax

%TimeNull

Description

Use this meta-SQL when you want to specify a NULL for a time field. Only use this meta-SQL in INSERT or UPDATE clauses. Do *not* use this meta-SQL in a WHERE clause.

Note. This meta-SQL is not implemented for COBOL.

This meta-SQL resolves into a database specific SQL substring.

Database	Resolves into
Informix	empty string (’’)
DB2	NULLIF(CURRENT TIME, CURRENT TIME)
All others	NULL

Parameters

None.

%TimeOut

Syntax

%TimeOut(*time_col*)

Description

%TimeOut expands to a platform-specific SQL substring representing *time_col* in the SELECT clause of an SQL query.

Parameters

Parameter	Description
time_col	Specify a time column.

%TimePart

Syntax

%TimePart(*DTTM_Column*)

Description

%TimePart returns the time portion of the specified datetime column.

Note. This meta-SQL is not implemented for COBOL.

Considerations using %TimePart

Use %TimeOut meta-SQL when fetching:

```
%TimeOut(%TimePart(DTTM_COLUMN)) from some_table
```

If a literal is used as the parameter to %TimePart, it must be wrapped in %DateTimeIn:

```
insert into some_table values(%TimePart(%DateTimeIn('2001-01-01-12.34.56.789012')))
```

Parameters

Parameter	Description
DTTM_Column	Specify the datetime column you want just the time part returned.

%TrimSubstr

Syntax

%TrimSubstr(*source_str*, *start*, *length*)

Description

%TrimSubstr, like %Substring, expands to a substring of *source_str*, except that trailing blanks are removed from the substring.

Parameters

Parameter	Description
source_str	Specify the source string.
start	Specify the substring's beginning position. The first character of *source_str* is position 1.
length	Specify the length of the substring.

See Also

Chapter 2, Meta-SQL , %Substring , page 550.

%Truncate

Syntax

%Truncate(*expression*, *factor*)

Description

%Truncate truncates an expression to a specified scale before or after the decimal point.

Parameters

Parameter	Description
Expression	Is an arbitrary expression involving numeric constants and database columns.
Factor	An integer or bind variable in SQLExec PeopleCode. The range of a factor is -30 to +31, negative number truncates to left of the decimal point.

Example

```
%Truncate(10.337, 2) = 10.33

%Truncate(13.37, 0) = 13

%Truncate(19.337, -1) = 10

SQLExec("SELECT %Truncate(field_c1, :1) from RECORD_T", field_c2, &Result);
```

where *field_c1* and *field_c2* are two fields in the record.

Illegal case

You may get wrong results or runtime SQL errors when using the illegal case.

```
%Truncate(10.337, 2 + 1)
(factor can not be an expression)
```

%TruncateTable

Syntax

%TruncateTable(*table_name*)

Description

%TruncateTable deletes all the rows in a table.

Note. You must use a table name, not a record name, with this statement.

On *all* databases, the use of %TruncateTable causes an implicit commit. The rows deleted by this command, and any other pending database updates, will all be committed. If you want to postpone the commit until subsequent database updates have been successfully completed, use the SQL statement 'DELETE FROM *table_name*' or 'IMPORT REPLACE WITH NULL' instead of %TruncateTable(*table_name*).The advantage of using %TruncateTable is that its execution is faster than either of these. %TruncateTable is often used for removing rows from a work table or a temporary table.

If you're calling %TruncateTable from an Application Engine program step, you should commit after the step that immediately precedes the step containing the %TruncateTable. Also, %TruncationTable should never be used on a step that is executed multiple times within a loop. In general, it's best to use this construct early in your Application Engine program as an initialization task.In addition, you want to avoid using this meta-SQL when your Application Engine program is started from **CallAppEngine**.

If you use %TruncateTable with %Table, you must specify the full name of the table. For example:

```
%TruncateTable(%Table(BAS_ELIG_DBGFLD))
```

Example

```
%TruncateTable(PS_TEMP_TABLE)
```

%UpdatePairs

Syntax

%UpdatePairs(*recname [correlation_id]*)

Description

The **%UpdatePairs** construct produces a comma separated list of *fieldname* = '*fieldvalue*' phrases for each changed field of the given record. Input processing is applied to the values, that is:

- If the field is a date, a time, or a datetime, its value will automatically be wrapped in %Datein(), %TimeIn(), or %DateTimeIn(), respectively.
- If the field is a string, its value will automatically be wrapped in quotes.
- If the field has a null value, NULL will be the given value.

Note. This meta-SQL can only be used in PeopleCode programs, not in Application Engine PeopleCode actions. Also, this meta-SQL is not implemented for COBOL.

Parameters

Parameter	Description
recname	Specify the name of the record to be used for updating. This can be a bind variable, a record object, or a record name in the form *recname*. You *can't* specify a RECORD.*recname*, a record name in quotes, or a table name.
correlation_id	Identifies the single letter correlation ID to be used to relate the record specified by *recname* and its fields.

Example

The record &REC has one key: FNUM. The FCHAR field has changed. Therefore, the following code:

```
Local record &REC;

&REC = CreateRecord(RECORD.MYRECORD);
&REC.FNUM.Value = 27;
&REC.FCHAR.Value = 'Y';
SQLExec("Update TABLE set %UpdatePairs(:1) where %KeyEqual(:1)", &REC)
```

expands to:

```
"Update TABLE set FCHAR = 'Y' where FNUM = 27"
```

The following example updates all the fields on a base record (&REC) that are *not* also fields on the related language record (&REC_RELATED_LANG). It creates a "holding" record (&REC_TEMP), copies the specific fields you want to update from the base record to the holding record, then uses the holding record for the update.

```
&UPDATE = CreateSQL("Update %Table(:1) set %UpdatePairs(:1) Where %KeyEqual(:2)");
&REC_TEMP = CreateRecord(@("RECORD." | &REC.Name));
&FIELD_LIST_ARRAY = CreateArray();
For &I = 1 to &REC_RELATED_LANG.FieldCount
   &FIELD_LIST_ARRAY.Push(&REC_RELATED_LANG.GetField(&I).Name);
End-For;

For &I = 1 to &REC.FieldCount
   If &FIELD_LIST_ARRAY.Find(&REC.GetField(&I).Name) = 0 then
      &REC_TEMP.GetField(&I).Value = &REC.GetField(&I).Value;
   End-If;
End-For;

&UPDATE.Execute(&REC_TEMP, &REC);
```

%Upper

Syntax

%Upper(*charstring*)

Description

The **%Upper** construct converts the string *charstring* to uppercase. You can use wildcards with *charstring*, like %.

Note. This meta-SQL is not implemented for COBOL.

Considerations With COBOL and Unicode

COBOL's uppercase function is not Unicode-aware, and corrupts Unicode data. If you want to use an uppercase function with COBOL, use the function supplied with PeopleTools called PTPUPPER.

The syntax to call PTPUPPER is:

```
CALL 'PTPUPPER' USING SQLRT

   <any PIC S9(4) COMP field that contains the fields
defined length (non-unicode)>

   <the String field - max PIC X(8192).>
```

The following is an example (from Unicode expanded source code):

```
01  W-WORK.

   02  W-DESCR        PIC X(90)  VALUE SPACES.
   02  W-SIZE         PIC S9(4) COMP VALUE +30.
   CALL 'PTPUPPER' USING SQLRT
```

```
W-SIZE OF W-WORK
W-DESCR OF W-WORK
```

Parameters

Parameter	Description
charstring	The string you want converted to uppercase.

Example

```
SELECT EMPLID, NAME FROM PS_EMPLOYEES WHERE %UPPER(NAME) LIKE %UPPER(sch%)
```

Meta-SQL Shortcuts

The following shortcuts are provided for when you want use the entire list of key fields for a record.

Note. The meta-SQL shortcuts can only be used in PeopleCode programs, not in Application Engine PeopleCode actions. Also, none of the meta-SQL shortcuts are implemented for COBOL.

%Delete

Syntax

`%Delete(:num)`

Description

This is a shorthand for:

```
Delete from %Table(:num) where %KeyEqual(:num)
```

%Insert

Syntax

`%Insert(:num)`

Description

This is a shorthand for:

```
Insert into %Table(:num) (%List(Nonnull_Fields :num)) values (%InsertValues(:num))
```

%SelectAll

Syntax

`%SelectAll(:num [ correlation _id])`

Description

%SelectAll is shorthand for selecting all fields in the specified record, wrapping date /time fields with %DateOut, %TimeOut, and so on.

The pseudo-code looks like this:

```
Select (All Fields, :num correlation_id) from %Table(:num) prefix
```

This shortcut is only appropriate if the statement is being used in PeopleCode or Application Engine to read data into memory. Dynamic Views should retain the internal database formats for DateTime fields.

Using SelectAll with CreateSQL

You can use SelectAll with the CreateSQL function without a record object. It must subsequently be executed with the record object with which you want to do the select. So for example you could do the following in your PeopleCode:

```
  &REC_PROJ_FUNDING = CreateRecord(Record.PROJ_FUNDING); /* free standing record⇒
object */
   /* Create SQL objects */
   &SQL_PROJ_FUNDING_SEL = CreateSQL("%SelectAll(:1)" /* bind this later */);
   &SQL_PROJ_FUNDING_UPDT = CreateSQL("%Update(:1)");
   /* Populate the record object */
   &REC_PROJ_FUNDING.BUSINESS_UNIT.Value = &BU_PC;
   &REC_PROJ_FUNDING.PROJECT_ID.Value = FetchValue(Scroll.GM_PROP_PROJ_VW, &I,⇒
DERIVED_GM_PROJ.PROJECT_ID);
   &REC_PROJ_FUNDING.ANALYSIS_GROUP.Value = &ANAL_GROUP;
   &REC_PROJ_FUNDING.TRANS_DT.Value = %Date;
   &SQL_PROJ_FUNDING_SEL.Execute(&REC_PROJ_FUNDING); /* bind the %SelectAll */
   While &SQL_PROJ_FUNDING_SEL.Fetch(&REC_PROJ_FUNDING);
```

You could also move the CreateRecord SQL statements out of the loop if you wanted (and then move the closes out of the loop too).

%SelectDistinct

Syntax

```
%SelectDistinct(:num [ prefix])
```

Description

%SelectDistinct is shorthand for selecting all fields in the specified record, wrapping date/time fields with %DateOut, %TimeOut, and so on.

The pseudo-code looks like this:

```
Select DISTINCT (All Fields, :num correlation_id) from %Table(:num) prefix
```

This shortcut is only appropriate if the statement is being used in PeopleCode or Application Engine to read data into memory. Dynamic Views should retain the internal database formats for DateTime fields.

%SelectByKey

Syntax

```
%SelectByKey(:num [ correlation_id ])
```

Description

This is a shorthand for:

```
Select %List(Select_List, :num correlation_id) from %Table(:num) correlation_id⇒
 where %KeyEqual(:num, correlation_id)
```

%SelectByKeyEffDt

Syntax

```
%SelectByKeyEffDt(:num1, :num2)
```

Description

This is a shorthand for:

```
Select %List(Select_List, :num1) from %Table(:num1) A where %KeyEqualNoEffDt(:num1⇒
 A) and %EffDtCheck(:num1 B, A, :num2)
```

%Update

Syntax

```
%Update(:num [ , :num2 ])
```

Description

This is a shorthand for:

```
Update %Table(:num) set %UpdatePairs(:num) where %KeyEqual(:num2)
```

If :*num2* is omitted, it defaults to :*num*.

CHAPTER 3

System Variables

The chapter provides an overview of system variables and then discusses each variable in detail.

Understanding System Variables

PeopleTools provides a number of system variables that provide access to system information. System variables are prefixed with the '%' character, rather than the '&' character. You can use these system variables wherever you can use a constant, passing them as parameters to functions or assigning their values to fields or to temporary variables.

In this section, we discuss each system variable.

%AllowNotification

Description

Indicates whether the Allow Notification check box for the current role's workflow routing options is selected. This system variable returns a Boolean value: True if the check box is selected (notifications allowed), False otherwise.

See Also

PeopleTools Security, Understanding PeopleSoft Security

%AllowRecipientLookup

Description

Indicates whether the Allow Recipient Lookup check box for the current role's workflow routing options is selected. This system variable returns a Boolean value: True if the check box is selected (recipient lookup allowed), False otherwise.

See Also

PeopleTools Security, Understanding PeopleSoft Security

%AsOfDate

Description

Returns the as-of-date of the environment that the PeopleCode is running in. In most cases, this is the current date, but for Application Engine environments, it is the processing date of the Application Engine program.

%AuthenticationToken

Description

This system variable returns a single sign on authentication token for the user after SwitchUser is executed. For example, you can use this system variable to write a single sign on cookie to the http response after a new user is authenticated.

Note. This system variable returns a valid value only after SwitchUser executes successfully. The value of this system variable is the authentication token itself. The value of the AuthTokenDomain Request object property is the domain across which the authentication token is valid, set in the AuthTokenDomain configuration property in the configuration properties file.

See Also

Chapter 1, PeopleCode Built-in Functions, SwitchUser , page 454, Chapter 15, Internet Script Classes (iScript) , AuthTokenDomain, page 944.

%BPName

Description

%BPName is relevant when the user has accessed a page from a worklist entry. It returns a string containing the name of the Business Process for the worklist entry. It returns an empty string if the user didn't access the current page group from a worklist.

%ClientDate

Description

%ClientDate returns the current date for the current user, adjusted for the user's time zone. This is potentially one day different than the server date. This is the date as specified with the current user's personalizations.

See Also

PeopleTools Security, Managing PeopleSoft Personalizations.

%ClientTimeZone

Description

%ClientTimeZone returns the current time zone for the current user as a three-character string. This is the timezone as specified with the current user's personalizations.

See Also

PeopleTools Security, Managing PeopleSoft Personalizations.

%Component

Description

%Component returns an uppercase character string containing the name of the current component, as set in the component definition.

%CompIntfcName

Description

%CompIntfcName returns the name of the Component Interface, if the currently executing PeopleCode program is being run from a Component Interface. If the currently executing PeopleCode program is *not* being run from a Component Interface, this variable returns NULL (if the program is running from PeopleCode) or "Nothing" (if running from Visual Basic.)

If you use this system variable in a mobile application, %CompIntfcName returns the name of the synchronizable Component Interface represented by the mobile object. An empty string is returned if no Component Interface is active.

%ContentID

Description

%ContentID returns the identification of the content for the current context as a string. The format of the value depends on the type of content.

PIA Content Type	Content ID Format
Component	*Menu.Component.Market*/**?Page**=*Page* **&Action**=*Action* &*Key ID = Key Value …*
Script	*Record.Event.Function*/?&*Parm ID = Parm Value …*
External	URL
Homepage	tab name
Template	template name
Query	query name
Worklist	worklist name
Navigation	Business Process Map name
File	file name

%ContentType

Description

%ContentType returns the type of content for the current content as a string.

For example, suppose your PeopleCode is part of the page USERMAIN_SELF, in this URL:

```
http://serverx/servlets/psp/eprocurement/hrms/c/MAINTAINT_SECURITY.USERMAIN_SEF.GBL
```

This system variable returns the following:

```
c
```

The content types are:

PIA Content Type	Value
Component	c
Script	s
External	e
Homepage	h
Template	t
Query	q
Worklist	w
Navigation	n
File	f

%Copyright

Description

This system variable returns a string suitable for use as a standard PeopleSoft copyright notice.

%Currency

Description

This system variable returns the preferred currency for the current user.

%Date

Description

%Date returns a Date value equal to the current server date.

%DateTime

Description

%DateTime returns the current server date and time as a Datetime value.

%DbName

Description

%DbName returns the name of the current database as a String value.

%DbServerName

Description

%DbServerName returns the name of the current Sybase or Informix database server as a string. This is not valid for other database types.

%DbType

Description

%DbType returns a string representing the type of the current database. The valid values are:

- APPSERVER
- DB2
- DB2UNIX
- INFORMIX
- MICROSFT
- ORACLE
- SYBASE

Note. Supported database platforms are subject to change.

%DeviceType

Description

This system variable returns the type of the mobile device. The values are:

- %MobileDevice_Laptop
- %MobileDevice_PDA
- %OtherDevice

%EmailAddress

Description

This system variable returns the email address of the current user.

%EmployeeId

Description

%EmployeeId returns an uppercase character string containing the Employee ID of the user currently logged on. This is typically used to restrict access to an employee's own records.

%ExternalAuthInfo

Description

This system variable returns external connect information. Programmers can customize the authentication process by passing in binary data. This data is encoded with base64 encoding and passed to sign on PeopleCode as a string using this system variable.

Note. This system variable can be used only in Signon PeopleCode. This system variable isn't applicable with the PeopleSoft Internet Architecture.

%FilePath

Description

This meta-variable returns the current file path as a string.

Note. This is *not* a system variable. This is a meta-variable only available in a Peoplesoft Application Engine program.

See Also

PeopleSoft Application Engine, Managing Application Engine Programs, Using the Command Line

%HPTabName

Description

This system variable returns the name of the last homepage tab visited by the user as a string.

%Import

Description

%Import returns True if an import is being performed by PeopleSoft Import Manager and False if not.

%IsMultiLanguageEnabled

Description

This system variable returns True if the current user is multi-language enabled.

See Also

PeopleTools Global Technology, Working With Language-Sensitive Application Data.
PeopleTools Global Technology, Working With Language-Sensitive Application Data

%Language

Description

%Language returns a string value representing the current setting from the Language menu if there is one, otherwise it returns the user's language setting (from PeopleSoft Security Administrator or user definition). It does not use the Language setting from the PeopleSoft Configuration Manager page. This value can be changed with the SetLanguage function.

See Also

Chapter 1, PeopleCode Built-in Functions, SetLanguage , page 420.

%Language_Base

Description

%Language_Base returns the base language for the current database, as set with the PeopleTools Options page.

See Also

PeopleSoft Server Tools Administration, PeopleTools Utilities, PeopleTools Options

%LocalNode

Description

%LocalNode returns the name of the local node for the current database as a string.

For example, suppose your PeopleCode is part of the page USERMAIN_SELF, in this URL:

```
http://serverx/servlets/psp/eprocurement/hrms/c/MAINTAINT_SECURITY.USERMAIN_SEF.GBL
```

This system variable returns the following:

```
hrms
```

See Also

PeopleSoft Integration Broker, Administering Basic Integrations

%Market

Description

The %Market system variable returns a three-character String value for the Market property of the current page group. This is useful if you want to add market-specific PeopleCode functionality to a component. For example:

```
if %Component = COMPONENT.PERSONAL_DATA then
   /* do some stuff that applies to all localized version */
   :
   :
   /* do some stuff that differs by market */
   evaluate %Market
      when = "USA"
       /* do usa stuff */
         break;
      when = "GER"
         /* do german stuff */
   end-evaluate;
end-if;
```

The Market property of a component specifies a component's target market. This property is set when a component is initially saved or cloned.

Components that are used on a global basis have a market setting of "GBL". Variations of components targeted at a specific market can have a local Market setting, for example "FRA". This enables developers to avoid cloning, renaming, and coding distinct PeopleCode in market-specific components. Instead, they can create a single component with market-specific PeopleCode, then clone the component, applying different Market property settings.

Because the %Market string is a three-character string like Country Code, Country Codes can be used as market settings where appropriate.

See Also

PeopleSoft Application Designer, Creating Component Definitions.

%MaxInterlinkSize

Description

%MaxInterlinkSize returns the current size limit of Interlink objects, as set on the PeopleTools Options page.

See Also

PeopleSoft Server Tools Administration, PeopleTools Utilities, PeopleTools Options

%MaxMessageSize

Description

%MaxMessageSize returns the current size limit of messages, as set on the PeopleTools Options page.

See Also

PeopleSoft Server Tools Administration, PeopleTools Utilities, PeopleTools Options

%Menu

Description

%Menu returns an uppercase string containing the current menu name. It can be used to restrict edits or processing to a specific menu.

%MobilePage

Description

This system variable returns the name of the current mobile page.

Example

```
%ThisMobileObject.QE_MB_RESULTS = %MobilePage;
```

%Mode

Description

%Mode returns a String value consisting of an uppercase character specifying the action a user selected when starting the current component. The following values can be returned. You can check either for the string value ("A", "U", and so on.) or for the constant:

Numeric Value	Constant Value	Description
A	%Action_Add	Add
U	%Action_UpdateDisplay	Update/Display
L	%Action_UpdateDisplayAll	Update/Display All
%Action_Correction	%Action_Correction	Correction
E	%Action_DataEntry	Data Entry
P	%Action_Prompt	Prompt

Note. This system variable is not valid with mobile PeopleCode.

%NavigatorHomePermissionList

Description

This system variable returns the navigator homepage permission list for the current user.

%Node

Description

%Node returns the name of the node from the current request object.

See Also

PeopleSoft Integration Broker, Administering Basic Integrations

%OperatorClass

Description

This system variable returns a string representing the primary or base class of the current operator.

Note. This system variable is supported for compatibility with previous releases of PeopleTools. New applications should use **%PermissionLists** instead.

See Also

Chapter 3, System Variables , %PermissionLists , page 575.

%OperatorId

Description

%OperatorId returns an uppercase character string containing the operator currently logged on. This is typically used to restrict access to records or fields to specific operators.

Note. This system variable is supported for compatibility with previous releases of PeopleTools. New applications should use **%UserId** instead.

See Also

Chapter 3, System Variables , %UserId , page 581.

%OperatorRowLevelSecurityClass

Description

This system variable returns a string representing the row-level security class of the current operator. The row-level security class is now distinct from the operator's primary class.

Note. This system variable is supported for compatibility with previous releases of PeopleTools. New applications should use **%RowSecurityPermissionList** instead.

See Also

Chapter 3, System Variables , %RowSecurityPermissionList , page 577.

%OutDestFormat

Description

This meta-variable returns the current output destination format as a string.

Note. This is *not* a system variable. This is a meta-variable only available in a Peoplesoft Application Engine program.

See Also

PeopleSoft Application Engine, Managing Application Engine Programs, Using the Command Line

%OutDestType

Description

This meta-variable returns the current output destination type as a string.

Note. This is *not* a system variable. This is a meta-variable only available in a Peoplesoft Application Engine program.

See Also

PeopleSoft Application Engine, Managing Application Engine Programs, Using the Command Line

%Page

Description

%Page returns an uppercase character string containing the current page name. It is typically used to restrict processing to a specific page, which is often necessary, because PeopleCode programs are associated with record definitions that can be shared by multiple pages.

%Panel

Description

%Panel returns an uppercase character string containing the current panel name.

Note. This system variable is supported for compatibility with previous releases of PeopleTools. New applications should use **%Page** instead.

See Also

Chapter 3, System Variables , %Page , page 573.

%PanelGroup

Description

%PanelGroup returns an uppercase character string containing the name of the current component, as set in the component definition.

Note. This system variable is supported for compatibility with previous releases of PeopleTools. New applications should use **%Component** instead.

See Also

Chapter 3, System Variables , %Component , page 564.

%PasswordExpired

Description

This system variable returns a Boolean indicating if the current user's password has expired. This system variable should be used after using SwitchUser, to verify if the password of the user that the user has just switched to is expired.

See Also

Chapter 1, PeopleCode Built-in Functions, SwitchUser , page 454.

%PerfTime

Description

Use the **%PerfTime** system variable to return the application server's *local* system time.

This variable returns only the local system time. This is different from the **%Time** system variable, which returns the system time from the database server, which may or may not be the same physical system as the application server.

PeopleSoft recommends using %PerfTime when measuring performance time for a specific PeopleCode program. This can enable developers to evaluate which coding logic has better performance time.

Note. Do not assume that %PerfTime returns the same time as the database server. Use %Time if you need to use a time value for your application transaction.

Example

The following is an example of how to use %PerfTime to check performance of a PeopleCode program:

```
&startTime = %PerfTime;

Local number &nbr;
Local Rowset &Table1_rs, &Table2_rs, &Table1_cpy_rs, &Table2_cpy_rs;
Local Rowset &Table1_vw_rs;


&Table1_rs = CreateRowset(Record.PTP_TABLE1);
&Table1_cpy_rs = CreateRowset(Record.PTP_TABLE1);
&Table1_rs.Fill("WHERE PTP_SEQ_NBR <= 10001");

REM
REM  Copy using Rowset function from one RowSet to Another
REM;

&Table1_rs.CopyTo(&Table1_cpy_rs);

REM
REM USE ROWSET TO READ RESULTS FROM A JOIN WITH BIND VARIABLE
REM;
```

```
&nbr = 10001;
&Table1_vw_rs = CreateRowset(Record.PTP_TABLE1_VW);
&Table1_vw_rs.Fill("WHERE PTP_SEQ_NBR >= :1", &nbr);

REM
REM  END OF EXERCISE CODE FOR PERFORMANCE COLLECTOR
REM;


&Rs = GetRowset(Scroll.PTP_TABLE1);

&Rs.Flush();
&Rs.Select(Record.PTP_TABLE1, "WHERE PTP_SEQ_NBR <= 10005");


&timeTaken = %PerfTime - &startTime;
```

See Also

Chapter 3, System Variables , %Time , page 580

%PermissionLists

Description

This system variable returns an array object containing entries for all the permission lists to which the current user belongs.

%Portal

Description

%Portal returns the name of the portal the current service is being accessed through, as a string. For example, suppose your PeopleCode is part of the page USERMAIN_SELF, in this URL:

```
http://serverx/servlets/psp/eprocurement/hrms/c/MAINTAINT_SECURITY.USERMAIN_SEF.GBL
```

This system variable returns the following:

```
eprocurement
```

See Also

PeopleSoft Integration Broker, Administering Basic Integrations

%PrimaryPermissionList

Description

This system variable returns a string representing the primary permission list of the current user.

%ProcessProfilePermissionList

Description

This system variable returns the process profile Permission List for the current user.

%PSAuthResult

Description

This system variable returns the result (True of False) of PeopleSoft ID and password authentication for the user signing on.

%Request

Description

%Request returns a reference to the request object. This reference can be used like an object, that is, you can use this as part of a dot notation string. For example:

```
&LOGOUT = %Request.LogoutURL;
```

This constant is applicable only in an internet script.

See Also

Chapter 15, Internet Script Classes (iScript) , page 931.

%Response

Description

%Response returns a reference to the response object. This reference can be used like an object, that is, you can use this as part of a dot notation string. For example:

```
&CookieArray = %Response.CookieNames();
```

This constant is applicable only in an internet script.

See Also

Chapter 15, Internet Script Classes (iScript) , page 931.

%ResultDocument

Description

This system variable returns a string containing an HTML document displayed to a user. This system variable is used with SwitchUser to pass any messages from the sign on process (or Signon PeopleCode) to the user.

Note. This system variable can be used only in Signon PeopleCode.

See Also

Chapter 1, PeopleCode Built-in Functions, SwitchUser , page 454.

%Roles

Description

This system variable returns an array object containing entries for all the roles to which the current user belongs.

%RowSecurityPermissionList

Description

This system variable returns a string representing the row-level PermissionList of the current user. The row-level security PermissionList is distinct from the user's primary PermissionList.

%RunningInPortal

Description

This system variable returns a Boolean value, letting you know if you're in the portal or not. This variable works in both frame templates and HTML templates.

%ServerTimeZone

Description

%ServerTimeZone returns the current time zone on the server as a three-character string.

%Session

Description

%Session returns a reference to the current, existing session. If you use %Session successfully, you don't have to use the GetSession function and Connect method. If you don't have a current session, %Session returns NULL.

Example

```
Local ApiObject &MySession

&MySession = %Session;
If Not (&MySession) Then
   /* Application level error handling */
End-If;
```

The following example is used with in a mobile application:

```
Local ApiObject &ob, &Session;
Local ApiObject &PropertyInfo;

&ob = %ThisMobileObject;
&Session = %Session;

&SaveCustomer = %ThisMobileObject.MyFavoriteCustomer;
```

```
&Customer = %Session.GetCompIntfc(COMPINTFC.CUSTOMER);
&Customer.Name="Acme Applicances";
```

%SignonUserId

Description

%SignonUserId returns the value the user typed in at the sign on page.

Note. This system variable can be used only in Signon PeopleCode.

%SignOnUserPswd

Description

%SignOnUserPswd returns the value the user typed in at the sign on page. This value is encrypted. This ensures end-user passwords can’t be "captured" by a Signon PeopleCode program.

Note. This system variable can be used only in Signon PeopleCode.

%SMTPBlackberryReplyTo

Description

This system variable returns the email address used by Blackberry to reply to, as a string, based on value in the application server configuration file for SMTPBlackberryReplyTo. This value is used in the Blackberry Response processing when Notification Templates are used.

See Also

PeopleSoft Workflow, Designing BlackBerry Email Responses

%SMTPGuaranteed

Description

This system variable returns a Boolean value, based on value in the application server configuration file for SMTPGuaranteed. The values are:

Value in Configuration File	Value Returned by System Variable
0	False
1	True

When this value is set to True, the SendMail built-in function and the Notification Send method send emails asynchronously by publishing an Application Message (EMAIL_MSG).

When this value is set to False, the SendMail built-in function and the Notification Send method send emails synchronously by calling the SMTP server directly.

See Also

Chapter 1, PeopleCode Built-in Functions, SendMail , page 394, Chapter 19, Notification Classes , Send, page 1067.
PeopleSoft Workflow, Designing BlackBerry Email Responses

%SMTPSender

Description

This system variable returns an email address as a string. The value is based on the value in the application server configuration file for SMTPSender. This value is used as the default sender email address for the following emails:

- TriggerBusinessEvent function
- SendMail function
- Notification class Send method

When using Notification Templates, if the Sender value is set to "System", this is the email address that is used for the Sender.

See Also

PeopleSoft Workflow, Designing BlackBerry Email Responses
PeopleSoft Workflow, Designing BlackBerry Email Responses

%SQLRows

Description

%SQLRows returns the number of rows affected by the most recent UPDATE, DELETE, or INSERT executed through **SQLExec**.

%SQLRows can also be used after SELECT. It returns 0 if no rows are returned, a non-zero value if one or more rows are returned. In this case, the non-zero value does not indicate the total number of rows returned.

%SyncServer

Description

This system variable returns a reference to the SyncServer object. If you don’t have a current SyncServer object, %SyncServer returns NULL.

See Also

Chapter 32, SyncServer Class, page 1633

%ThisMobileObject

Description

Returns a reference to the current mobile object, which is the equivalent of a row for a Component Interface object.

To get the top level object, use %ThisMobileObject.GetTopParent.

To get the immediate parent, use %ThisMobileObject.GetParent.

Example

```
Local ApiObject &ob, &Session;
Local ApiObject &PropertyInfo;

&ob = %ThisMobileObject;
&Session = %Session;

/*********************************************************
* Set properties directly on %ThisMobileObject           *
*********************************************************/
%ThisMobileObject.date  = "2002.12.06";
%ThisMobileObject.time  = "10:30:00";
```

See Also

Chapter 18, Mobile Classes , page 1009

%Time

Description

%Time retrieves the current database server time.

If your application deals with time-sensitive data, use this value. If you want to measure the performance of a PeopleCode program, use the %PerfTime system variable instead.

See Also

Chapter 3, System Variables , %PerfTime, page 574.

%TransformData

Description

This system variable returns a reference to the TransformData object. If you don’t have a current TransformData object, %TransformData returns Null.

See Also

PeopleSoft Integration Broker, Applying Filtering, Transformation, and Translation

%UserDescription

Description

This system variable returns the description (if any) listed for the current user.

%UserId

Description

%UserId returns a character string containing the user currently logged on. This is typically used to restrict access to records or fields to specific users.

%WLInstanceID

Description

%WLInstanceID returns a string containing the name of the Worklist Instance ID for the current worklist entry. It returns a blank string if the current page was not accessed using a worklist.

%WLName

Description

%WLName returns a string containing the name of the Worklist for the current worklist entry. It returns a blank string if the current page was not accessed using a worklist.

CHAPTER 4

Meta-HTML

This chapter provides an overview of Meta-HTML and discusses the following topics:

- Variables
- Functions
- Comments
- Alphabetical list of Meta-HTML elements

Understanding Meta-HTML

PeopleSoft Internet Architecture page processing includes functionality to perform certain substitutions on the generated HTML. These substitutions are known as Meta-HTML. These Meta-HTML elements enable access to some of the environment, and in some cases, to perform browser-dependent substitutions.

The Meta-HTML processing is performed on the entire page, including the contents of any HTML areas in the page. Thus Meta-HTML can be used in an HTML area.

Meta-HTML processing is not currently done in the results of an internet script, so the iScript programmer cannot use Meta-HTML.

A limited subset of the Meta-HTML processing is also done on any JavaScript or auxiliary HTML files attached and downloaded to the web server. This processing occurs both for files attached to PeopleSoft Internet Architecture pages and for files attached to an iScript.

An auxiliary file may be attached to an HTML area using the Meta-HTML %JavaScript method.

An auxiliary page may be attached to a iScript page or a PeopleSoft Internet Architecture page using the Response method GetJavaScriptURL.

Each Meta-HTML element that may be used in auxiliary files is noted in its description.

Find References Tool Considerations

When you specify a definition name in an HTML area, it is *not* found using the Find References tool. It also won't be automatically renamed when a definition is renamed. All text within an HTML area is treated like a quoted string, a literal.

For example, Find References won't find the image PSLOGO or the HTML definition PT_EDITSCRIPTS.

```
<img src='%Image(PSLOGO)'>

<script src='%JavaScript(PT_EDITSCRIPTS)'></script>
```

See Also

Chapter 4, Meta-HTML, %JavaScript , page 587

Chapter 15, Internet Script Classes (iScript) , GetJavaScriptURL, page 952

Variables

A Meta-HTML variable has the form %name, similar to the PeopleCode system variables. It's replaced by the substituted value wherever it appears. In the following example, the Meta-HTML variable is replaced by the current component name:

```
%Component
```

Functions

A Meta-HTML function has the following form:

```
%name(parameter, parameter...)
```

The entire expression is replaced by a substituted value, where the parameters are used in determining the value to be substituted. The parameters are arbitrary sequences of characters separated by commas. Do *not* place quotes around the parameters unless they form part of the value to be used.

In the following example, the entire text is replaced by the contents of the message 126, 45 from the message catalog, or the phrase "Unable to load images" if that message isn't found.

```
%Message(126, 45, Unable to load images)
```

Comments

The Meta-HTML processor recognizes two forms of comments. These comments are deleted from the generated HTML or JavaScript. They enable the application developer to comment the HTML objects in the database without increasing the size of the HTML passed to the browser.

The recognized comments are as follows:

Two slashes followed by a percent sign at the start of a line deletes everything to the end of the line containing the slashes.

```
//%  anything
```

A less-than sign, followed by an exclamation mark and a percentage sign deletes everything from those marks to the mark -->, which may be on another line.

```
<!%  anything   -->
```

These Meta-HTML comments may be used both in HTML areas and attached auxiliary files.

Alphabetical List of Meta-HTML Elements

In this section, we discuss each Meta-HTML element. Throughout this section, typographical conventions are used to distinguish between different elements of the PeopleCode language, such as bold to indicate function names, italics for arguments, and so on.

See Also

Chapter 1, PeopleCode Built-in Functions, PeopleCode Typographical Conventions, page 1.

%Appserver

Description

At runtime %Appserver is replaced with the name of the application server.

This metavariable is valid for use in attached auxiliary files.

%AppsRel

Description

At runtime %AppsRel is replaced with the application release string.

This metavariable is valid for use in attached auxiliary files.

%Browser

Description

At runtime %Browser is replaced with the browser name as specified by the browser loading the current page.

%BrowserPlatform

Description

At runtime %BrowserPlatform is replaced with the operating system name as specified by the browser loading the current page.

%BrowserVersion

Description

At runtime %BrowserVersion is replaced by the version string as specified by the browser loading the current page.

%Cols

Syntax

%Cols(*n*)

Description

At runtime %Cols(*n*) is replaced with COLS=*n*.

Note. For Netscape 4.x, this Meta-HTML function generates nothing (it's suppressed). This is used as a workaround for some Netscape considerations.

Parameters

Parameter	Description
n	Specify the number of columns.

This metafunction is valid for use in attached auxiliary files, but always generates COLS=n, that is, it isn't Browser-aware when used in attached files.

%Component

Description

At runtime %Component is replaced with the component name of the current component.

%Copyright

Description

At runtime %Copyright is replaced with a string suitable for use as a standard PeopleSoft copyright notice.

This metavariable is valid for use in attached auxiliary files.

%DBName

Description

At runtime %DBName is replaced with the name of the application database.

This metavariable is valid for use in attached auxiliary files.

%DBType

Description

At runtime %DBType is replaced with the type of the application database.

This metavariable is valid for use in attached auxiliary files.

%Encode

Syntax

%Encode(*anything*)

Description

At runtime %Encode plus *anything* is replaced with the encoded string. Encoding is done according to normal URL encoding rules.

Parameters

Parameter	Description
anything	Specify the string to be encoded.

This metafunction is valid for use in attached auxiliary files.

%Formname

Description

At runtime %Formname is replaced with the name of the HTML FORM generated for the current PIA page.

This metavariable is valid for use in attached auxiliary files.

%Image

Syntax

%Image(*imagename*)

Description

At runtime %Image and *imagename* are replaced by the URL suitable for referencing the image on the web server. In addition, the image is loaded into the web server's cache directory.

Parameters

Parameter	Description
imagename	Specify the name of an image saved as an image definition in PeopleSoft Application Designer.

Example

```
<img src='%Image(PSLOGO)'>
```

%JavaScript

Syntax

%JavaScript(*HTMLDefinition*)

Description

At runtime %JavaScript and the *HTMLDefinition* are replaced by the URL suitable for referencing the .js file on the web server. In addition, the JavaScript is loaded into the web server's cache directory.

Parameters

Parameter	Description
HTMLDefinition	Specify the name of an HTML definition that contains a JavaScript program.

Example

```
<script src='%JavaScript(PT_EDITSCRIPTS)'></script>
```

%LabelTag

Description

At runtime %LableTag is replaced with the text LABEL.

Note. For Netscape 4.x, %LabelTag generates SPAN. This is used as a workaround for some Netscape considerations with the unimplemented LABEL tag.

This metavariable is valid for use in attached auxiliary files, but always generates LABEL, that is, it isn't Browser-aware when used in attached files.

%Menu

Description

At runtime %Menu is replaced by the menu name for the currently loaded component.

%Message

Syntax

%Message(*message_set*, *message_num*, *default_msg_txt* [, *paramlist*])

where *paramlist* is an arbitrary-length list of parameters to be substituted in the resulting text string, in the form: *param1* [, *param2*]. . .

Description

%Message retrieves a message from the PeopleCode Message Catalog and substitutes in the values of the parameters into the message.

The *message_set* and *message_num* parameters specify the message to retrieve from the catalog. If the message is not found in the Message Catalog, the default message provided in *default_msg_txt* is used. Message sets 1 through 19,999 are reserved for use by PeopleSoft applications. Message sets 20,000 through 32,767 can be used by PeopleSoft users.

The parameters listed in the optional *paramlist* are referenced in the message using the % character followed by an integer referencing the position of the parameter in the function call. For example, if the first and second parameters in *paramlist* were MONDAY and 12/5/2001, they would be inserted into the message string as %1 and %2. To include a literal percent sign in the string, use %%; %\ is used to indicate an end-of-string and terminates the string at that point, this is generally used to specify fixed-length strings with trailing blanks.

The message is obtained using the current user language code.

Parameters

Parameter	Description
message_set	Specify the message set to be retrieved from the catalog. This parameter takes a number value.
message_num	Specify the message number to be retrieved from the catalog. This parameter takes a number value.
default_msg_txt	Specify the text to be displayed if the message isn't found. This parameter takes a string value.
paramlist	Specify values to be substituted into the message.

This is valid for use in attached auxiliary files.

Note. If the message is changed (or a new language version is added) after the auxiliary file has been loaded to the web server, the auxiliary file still contains the old version of the message. It is necessary to manually delete the file from the web server cache directory to get it to re-retrieve the (unmodified) auxiliary component with the (modified) message bindings.

%Page

Description

At runtime %Page is replaced by the name of the current page.

%ServicePack

Description

At runtime %ServicePack is replaced with the application service pack string.

This metavariable is valid for use in attached auxiliary files.

%SubmitScriptName

Description

At runtime %SubmitScriptName is replaced with the name of the JavaScript function that the current PIA page uses to submit the form when a server action is required.

This metavariable is valid for use in attached auxiliary files.

%ToolsRel

Description

At runtime %ToolsRel is replaced with the tools release string.

This metavariable is valid for use in attached auxiliary files.

%URL

Syntax

%URL(*URLIdentifier* [, **NOENCODE** | **ENCODE** | **DESCR**])

Description

The %URL Meta-HTML function finds the URL specified by *URLIdentifier* and substitutes its value. The *URLIdentifier* must already exist and have been created using URL Maintenance.

Parameters

Parameter	Description
URLIdentifier	Specify a URL Identifier for a URL that already exists and was created using the URL Maintenance page.
NOENCODE | **ENCODE** | **DESCR**	Specify any encoding or other processing to be done with the URL. ENCODE is the default value. If you specify ENCODE, special characters in the URL are encoded using standard URL encoding rules, that is, blanks are replaced with %20, and so on. If you specify NOENCODE, no encoding is done with the URL. If you specify DESCR, the description from the URL definition is used instead of the URL itself.

This metafunction is valid for use in attached auxiliary files.

Example

```
<a href='%URL(homepage)'>%URL(homepage, DESCR)</a>
```

See Also

PeopleSoft Server Tools Administration, PeopleTools Utilities, URL Maintenance.

%UserId

Description

At runtime %UserID is replaced with the current UserId.

CHAPTER 5

AESection Class

This chapter provides an overview of AESection class and discusses the following topics:

- How an AESection is accessed.
- AESection example.
- Data type of an AESection object.
- Scope of an AESection object.
- AESection Class built-in function.
- AESection Class reference.

Understanding AESection Class

Before PeopleTools 8, users could perform SQL directly on the Application Engine tables, thereby changing their SQL and Application Engine "flow" in a dynamic manner, prior to running their applications. Some applications, for example, let the user input their "rules" in a user-friendly application, then convert these rules, at save time, into Application Engine constructs.

With PeopleTools 8, Application Engine programs should not perform SQL directly on the Application Engine tables, as they are system tables, and are cached. SQL on the Application Engine tables may not be accurately reflected when the applications are executed, because of the caching mechanism.

To overcome this problem, developers have two basic operations that let them modify their Application Engine programs from their online pages:

- Ability to modify SQL definitions, which are referenced within Application Engine SQL using the PeopleCode SQL class and the meta-SQL function %SQL.
- Ability to dynamically change the execution flow of a given Application Engine section.

Use the AESection class to do the latter. This section object is used to modify the steps and SQL associated with a given section by using PeopleCode.

Before describing the specifics of the AESection class, the following are some general terms to understand:

- *base* section

  The *base* section represents the Application Engine section you are working on. This is the section that you are changing in PeopleCode. In other words, it’s the target section.

- *template* section

The *template* section represents the Application Engine section that is the source, or "model" for the section you're building. You copy *from* the template section *to* the base section.

The template must exist in the database before you can use it.

If the base section you specify doesn't exist, a *new* base section is created. This enables you to dynamically create sections as needed.

You can copy steps (and their attributes) from the template to the base. The only attribute of the step you can modify is the SQL statement that gets executed.

Warning! When you open or get an AESection object, (that is, the base section) any existing steps in the section are deleted. You must add a new step to the section before you can modify it.

The main assumption for this class is that your rules are dynamic primarily in the SQL that they execute, but that the structure of the rules are static, or at least defined well enough that a standard template can be applied.

Note that you can't update the Application Engine *actions* that are not SQL-related (that is, PeopleCode, Call Section, or Log Message). In other words, you can't change the PeopleCode associated with a PeopleCode action within a step. You can add a step containing a PeopleCode action to your new section, but you can't change the PeopleCode dynamically.

See Also

Chapter 31, SQL Class , page 1613

Chapter 2, Meta-SQL , %SQL , page 548

PeopleSoft Application Engine, Creating Application Engine Programs, Adding Steps

How an AESection is Accessed

When an AESection is opened (or accessed), the system first checks if it exists with the given input parameters.

If the base section you specify doesn't exist, a *new* base section is created. This enables you to dynamically create sections as needed. In addition, if the target section doesn't exist, all section-level attributes are copied from the template to the target section. If the target exists, it retains its attribute settings.

If an effective date is specified (with **EffDt**), but there is no match using that effective date, the AESection is opened using the base effective date of '1900-01-01'.

The market defined for the current component is used for the section to be open. If no section with this market exists, the default GBL is used.

If you open an AESection object from within a running Application Engine program, the market value is set to the value of the current process.

The database platform you're currently running is used as the database platform for the section to be opened. If no section with this database platform exists, the default "none" is used.

To find the correct section to open, the precedence is:

1. Market
2. Database platform
3. Effective date

When a section is closed and its changes are saved, the market and database platform from the system that performed the changes is used as the market and database platform for the modified section.

Let's take an example. Assume that you're running on DB2, and your current market is set to 'MKT'. Assume that you want to open a section called Sect1 in an Application Engine program called TestAppl, and that your data looks something like this:

Number	Application Engine Program	Application Engine Section	Market	Platform	Effective Date
1	TestAppl	Sect1	USA	DB2	1990-01-01
2	TestAppl	Sect1	USA	None	1900-01-01
3	TestAppl	Sect1	GBL	None	1900-01-01

If you use:

```
&SECTION.Open("TestAppl", "Sect1", "1998-12-14");
```

Then the third section in the previous list is used, because the market takes highest precedence. When the section is saved, the values for market and platform are updated to the current system values.

Warning! When you open or get an AESection object, (that is, any base section) any existing steps in the section are deleted. You must add a new step to the section before you can modify it.

The other attributes of the section, however, are retained, with the exceptions noted previously.

AESection Example

Assume that you have a template section called TEMPLATE in the Application Engine program called MY_APPL. The template looks like this:

Steps	Actions
NewStep1	DO When DO Select SQL
NewStep2	DO Select CallSection

Also, assume that you have a base section called DYN_SECT in the Application Engine program called RULES. When you start, this section looks like this:

Steps	Actions
Step1	DO Select Call Section DO Unit
Step2	Call Section

Here's the PeopleCode:

```
Local AESection &Section;
&Section = GetAESection("RULES", "DYN_SECT");
/* Open the base section */
&Section.SetTemplate("MY_APPL", "TEMPLATE");
/* Set the template section */
&Section.AddStep("NewStep2");
/* Insert NewStep2     */
/* Do some SQL stuff here  */
&Section.SetSQL("DO_SELECT", &MySql);
/* Modify the SQL in the added step */
&Section.Save();
&Section.Close();
/* Save and close   */
```

The base section looks like this after execution:

Steps	Actions
NewStep2	DO Select Call Section

Note. The existing steps in the base section have been *overwritten* by the new step from the template section.

Data Type of an AESection Object

You should declare an AESection object as type AESection. For example:

```
Local AESection &SECTION;
```

Scope of an AESection Object

An AESection object can only be instantiated from PeopleCode.

The AESection Object is designed for use within online pages. Typically, dynamic sections should be constructed in response to an end-user action.

Note. Do not call an AESection object from an Application Engine PeopleCode Action. If you need to access another section, use CallSection Actions instead.

See Also

PeopleSoft Application Engine, Creating Application Engine Programs, Call Section Actions

AESection Class Built-in Function

Chapter 1, PeopleCode Built-in Functions, GetAESection , page 233

AESection Class Methods

In this section, we discuss each AESection class method, in alphabetical order.

AddStep

Syntax

AddStep(*ae_step_name* [, *NewStepName*])

Description

The given step name from the template section is added as the next step into the base section, and named the existing step name.

Note. When you open or get a section, all the existing steps are deleted. The first time you execute AddStep, you add the first step. The second time you execute AddStep, you add the second step, and so on.

All attributes of the step are copied, including all of its actions. The only changeable attribute of a step are its SQL statements, which can be modified using the SetSQL method. SetSQL is run on the *current* step, that is, the step most recently added.

You can also change the name of the step by using the optional parameter *NewStepName*.

If the step named does not exist in the template, an error occurs. Likewise, if you have not opened or set the template for the section, you receive an error.

Parameters

Parameter	Description
ae_step_name	Specify the step name from the template to be added as the *next* step in the base section. This parameter takes a string value.
NewStepName	Specify a new name for the step to be added. This parameter is optional, and takes a string value.

Returns

None.

Example

See Chapter 5, AESection Class , AESection Example , page 593.

See Also

Chapter 5, AESection Class , SetSQL, page 598.

Close

Syntax

```
Close()
```

Description

Close closes the AESection object. Any unsaved changes to the section are discarded.

Parameters

None.

Returns

None.

Example

See Chapter 5, AESection Class , AESection Example , page 593.

See Also

Chapter 5, AESection Class , Save, page 598, Chapter 5, AESection Class , Open, page 597, Chapter 1, PeopleCode Built-in Functions, GetAESection , page 233.

Open

Syntax

Open(*ae_applid*, *ae_section*, [*effdt*])

Description

The **Open** method associates the AESection object with the given Application Engine section, based on the *ae_applid* and *ae_section*. If the *effdt* is specified, this is also used to get the section object. In other words, the Open method sets the base section.

Warning! When you open or get an AESection object, (that is, the base section) any existing steps in the section are deleted.

Note. If the base section you specify doesn't exist, a *new* base section is created. This enables you to dynamically create sections as needed. In addition, if the target section doesn't exist, all section-level attributes are copied from the template to the target section. If the target exists, it retains its attribute settings.

If the AESection is still open when you issue the Open method, the previously opened object is discarded and none of the changes saved. To prevent accidentally discarding your changes, you can use the IsOpen property to verify if a section is already open.

The AESection is open based on the Market and database type of your current system.

Parameters

Parameter	Description
ae_applid	Specify the application ID of the section you want to access. This parameter takes a string value.
ae_section	Specify the section name of the section you want to access. This parameter takes a string value.
effdt	Specify the effective date of the section you want to access (optional). This parameter takes a string value.

Returns

An AESection object.

Example

```
Local AESection &SECTION;
```

```
&SECTION = GetAESection("RULES1", "DYN_SECT");

/* do some processing */

&SECTION.Close();

&SECTION.Open("RULES2", "DYN_SECT");

/* do additional processing */
```

See Also

Chapter 5, AESection Class , IsOpen, page 600, Chapter 5, AESection Class , How an AESection is Accessed , page 592, Chapter 1, PeopleCode Built-in Functions, GetAESection , page 233.

Save

Syntax

Save()

Description

The **Save** method saves the section to the database. The AESection object remains open after you use this method. To close the object, you must use the **Close** method.

You must commit all database changes prior to using this method. This is to avoid locking critical Tools tables and hence freezing all other users. You receive a runtime error message if you try to use this method when there are pending database updates, and your PeopleCode program terminates. You need to commit any database updates prior to using this method. Use the CommitWork PeopleCode function to commit database updates.

Parameters

None.

Returns

None.

Example

See Chapter 5, AESection Class , AESection Example , page 593.

See Also

Chapter 5, AESection Class , Close, page 596, Chapter 5, AESection Class , Open, page 597, Chapter 1, PeopleCode Built-in Functions, CommitWork , page 71.

SetSQL

Syntax

SetSQL(*action_type_string*, *string*)

Description

The **SetSQL** method replaces the SQL associated with the given action type in the current step in the base section with the SQL in *string*. The *current* step is the latest step that was added using AddStep. The action types are:

- DO_WHEN
- DO_WHILE
- DO_SELECT
- SQL
- DO_UNTIL

Note. All action types must be passed in as strings with quotation marks.

If the action specified does not exist in the current step, an error occurs.

You can use a SQL object as *string*.

```
&SECTION.SetSQL("SQL", &SQL);
```

Parameters

Parameter	Description
action_type	Specifies the action type of the current step that should be changed. This parameter takes a string value.
string	Specifies the SQL to be used to replace the SQL in the current step.

Returns

None.

Example

See Chapter 5, AESection Class , AESection Example , page 593.

See Also

Chapter 5, AESection Class , AddStep, page 595, Chapter 31, SQL Class , page 1613.

SetTemplate

Syntax

SetTemplate(*ae_applid, ae_section*)

Description

The **SetTemplate** method sets the **template** to be used with an AESection object, as identified by *ae_applid* and *ae_section*.

The rules for assigning a template section are similar to the rules for selecting a base section. The selection of market and database platform are done exactly the same. However, the effective date of the template is always set based on the effective date of ‘1900-01-01’.

You must set the template before you can use any of the other methods.

Parameters

Parameter	Description
ae_applid	Specify the name of the Application Engine program that contains the section to be used as the template.
ae_section	Specify the name of the Application Engine section in the program to be used as the section template.

Returns

None.

Example

See Chapter 5, AESection Class , AESection Example , page 593.

See Also

Chapter 5, AESection Class , Open, page 597, Chapter 5, AESection Class , How an AESection is Accessed , page 592, Chapter 1, PeopleCode Built-in Functions, GetAESection , page 233.

AESection Class Property

In this section, we discuss the IsOpen AESection property.

IsOpen

Description

If this property is True, the section object is already open. If you try to open a section object that is already open, the open section object is closed and all the changes that haven't been saved are discarded before the object is re-opened.

Example

```
If Not(&MYSECTION.IsOpen) Then
   &MYSECTION.Open(MYAPPLID, SECTION2);
End-If;
```

CHAPTER 6

Application Classes

This chapter provides an overview of application classes and discusses the following topics:

- Application classes general structure.
- Import declarations.
- Self-reference.
- Property overrides.
- Superclass reference.
- Downcasting.
- Exception handling.
- Design of base classes.
- Declaration of application classes.
- Scope of an application class.
- Application classes built-in functions and language constructs.

Understanding Application Classes

You can create Application Packages in PeopleSoft Application Designer. These *packages* contain application classes (and may contain other packages also). An application class, at its base level, is just a PeopleCode program. However, using the Application Packages, you can create your own classes, and extend the functionality of the existing PeopleCode classes.

The application classes provide capability beyond that offered by the existing PeopleCode classes. Unlike the existing classes, a subclass can inherit all the properties and methods of the class it extends. This provides all the advantages of true object-oriented programming:

- easier to debug because all the pieces are separate
- easier to maintain because functionality is gathered into a single place
- extensible by subclass

In addition, the application classes provide more structure. Using the Application Packages, you have a clear definition of each class, as well as its listed properties and methods. This makes it easier to create a complex program that uses many functions.

See Also

PeopleCode Developer's Guide, Introducing Application Packages

When Would You Use Application Classes?

Use Application classes to help structure your existing PeopleCode functions. For example, prior to Application classes, most PeopleCode functions were put into FUNCLIB records. However, in larger, more complex programs, it is sometimes difficult to find (let alone maintain) a function. You could make every function a class in a package with very little reworking of your PeopleCode (as well as making the different fields in a record a package, and combining them into a single larger package.) This provides a better visual interface for grouping your functions.

In addition, use Application classes when you have to do generic processing, and just the details are different. For example, suppose one of the processes for your application is reading a file object and doing bulk insert. The process could all be contained in a single package, with the classes and subclasses providing all the details, different kinds of reading for different types of files, different inserts based on the record, and so on.

One of the main differences between a class and a function call is that the call to the class is dynamic, like a function variable, but more closely controlled. All the calls to the class (methods) must have the same signature.

For example, suppose you want to provide a more generic sort, with comparison function at the end of it. You want to use the array class Sort method, but your process has to be generic: you aren't certain if you're comparing two records or two strings. You could define a class that had as its main method a comparison, and return a -1, 0, or 1. Then write your personalized sort, extending the array class method Sort.

Another use is for business processing. Think in terms of core functionality, with vertical product solutions that extend the basic processing by providing specifics, in other words, by extending the general classes with subclasses. This could be appropriate for sales force automation or order entry.

Generally, use application classes where you can extract the common functionality.

Application Classes General Structure

All application classes are contained in a *package*. All application classes are composed of PeopleCode. In this section, we discuss the general form of an application class, which is:

- Class name
- Class extensions
- Declaration of public external interface
- Declaration of private instance variables and methods
- Definition of methods
- Constructors

This division enables a separation of the following:

- what the class provides to other classes
- what is applicable to the entire class

• what is applicable to the definition of a method

The data types used in an application class (for methods parameters, return values, and so on) can be any PeopleCode types, including application classes. Likewise, application classes can be used anywhere in a PeopleCode program where a general data type can be used.

See Also

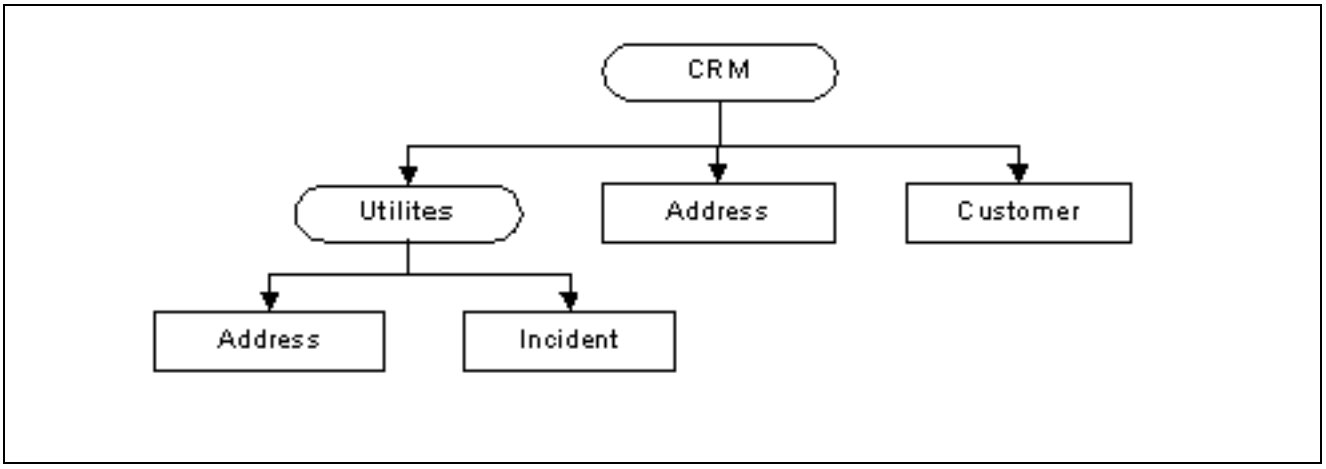
PeopleCode Developer's Guide, PeopleCode Language, Data Types

Class Name

Application classes have a fully qualified name that is formed hierarchically by the name of the top-level package that contains them, the package that contains that package, and so on down to the short class name, that is, the one specified when the class was created in PeopleSoft Application Designer, using a colon for the separator between names.

The fully qualified name of the class must be unique, as this is what identifies the class. The short name of the class does not have to be unique.

For example, suppose package CRM contains the application classes Address and Customer as well as the package Utilities, which in turn contains the application classes Address and Incident:



Application classes example

There are four distinct application classes in this example:

• CRM:Address

• CRM:Customer

• CRM:Utilities:Address

• CRM:Utilities:Incident

Note. If you change the name of a package or an application class, that name change is *not* automatically propagated through all your PeopleCode programs. You must make the change manually in your programs. If you specify an incorrect name for a package or application class, you receive a warning when you try to save the PeopleCode.

Class Extension

Extension represents the "is-a" relationship. When a class extends another class, it's called a *subclass* of that class. Conversely, the class being extended is called a *superclass* of that subclass.

A subclass inherits all of the public methods and properties (collectively called *members*) of the class it extends. These members can be overridden by declarations of methods and properties in the subclass.

Note. Application classes have no multiple inheritance (that is, being able to extend more than one class) or concept of interface (that is, a definition of a set of methods that one or more classes implement).

Type checking in PeopleCode (both at design time and runtime) does strong type checking of application classes, tracking each application class as a separate type. A subclass can be used as the class it extends, because it implements the public interfaces of its superclass. This is called *subtyping*.

In the following example, the class Banana extends the class Fruit. Banana is the *subclass* of Fruit. From Fruit, you can extend to Bananas. However, from Bananas you can't extend to Fruit. Another way to think about this is you can only extend from the general to the specific, not the other way around.

```
class Fruit
   method DoFruit();
   property number FruitNum instance;
end-class;

class Banana extends Fruit
   method DoBanana();
   property number BananaNum instance;
end-class;
```

The following code shows a correct way to assign a class object, because Banana is a *subtype* of Fruit.

```
local Fruit &Bref = Create Banana();
```

The following is *not* correct. Banana is a subtype of Fruit. Fruit is *not* a subtype of Banana. This assignment causes an error at design time.

```
local Banana &Fref = Create Fruit();
```

Before you can extend a class, you must first import it. You must import all class names you use in a PeopleCode program before you use them.

See Also

Chapter 6, Application Classes , Import Declarations , page 612

Declaration of Public External Interface

The names of the members (that is, all the methods and properties of a class) must be unique only within that class. Different classes can have members with the same name. As members are used only with objects, and each object knows its class, the member that's accessed is the one associated with that class.

For example, more than one class has the property Name which returns the name of the object executing the property. There isn't any confusion, because the field Name property returns the name of a field, while the record name property returns the name of a record. No class has more than one property called Name.

Note. Within a class, each member name *must* be unique.

The public part of a class declaration specifies the methods and properties that the class provides to other PeopleCode programs. These methods and properties are *dynamically bound*, that is, the actual method that is called depends on the actual class of the object, as it might be an overridden method or property from a subclass.

In the following code example, the text in bold indicates the public part of the class declaration.

```
/* generic building class */
class BuildingAsset
   method Acquire ();
   method DisasterPrep();
end-class;
```

Declaration of Private Instance Variables and Methods

The private part of a class declaration gives the declaration of any private methods, instance variables, and private class constants. Private methods *cannot* be overridden by subclasses, because they are completely private to the declaring class. Likewise, the instance variables can't be overridden.

The declarations of private variables and methods are done within the class declaration. In the following example, there are both public as well as private methods and properties. Any method that is declared before the keyword **Private** in the initial class declaration is public. Any method declared after the keyword **Private** is private. Properties are only declared in **Public**. Instances and Constants are only declared in **Private**

In the following example, the class Example extends the class ExampleBase. It has both public as well as private methods and properties. It also defines the private method in the definition of methods section of the code (the private definitions are marked in bold text.)

```
import PWWPACK:ExampleBase;

class Example extends ExampleBase
   method Example();
   method NumToStr(&Num As number) Returns string;
   method AppendSlash();
   property number SlashCount get;
   property number ImportantDayOfWeek get set;
   property string SlashString readonly;
   property date ImportantDate;
private
   method NextDayOfWeek(&Dow As number) Returns date;
   Constant &Sunday = 1;
   instance string &BaseString;
end-class;

Global string &CurrentBaseString;

/* Method declarations */
method NumToStr
   return String(&num);
```

```
end-method;

method AppendSlash
   &SlashString = &SlashString | "/";
end-method;

get SlashCount
   Return Len(&SlashString) - Len(&BaseString);
end-get;

get ImportantDayOfWeek
   Return Weekday(&ImportantDate);
end-get;

set ImportantDayOfWeek
   &importantdate = %This.nextdayofweek(&newvalue);
end-set;

/* Private method. */
method nextdayofweek
   Return &ImportantDate + (&dow - Weekday(&ImportantDate));
end-method;

/* Constructor. */
method Example
   &BaseString = &CurrentBaseString;
   &SlashString = &BaseString;
   &ImportantDate = Date(19970322);
end-method;
```

Definition of Methods

After the declaration of any global or component variables and external functions needed by the methods, comes the actual definition of the methods. This section discusses some of the differences in how application programs are processed by the system, and how application class method parameters are processed.

- The system *never* skips to the next top-level statement.
- Application programs generally pass parameters by value, which is not the same as for existing functions.
- Parameter passing with object data types is by reference.
- Application programs use the out specifier to pass a parameter by reference.

Not Skipping to The Next Top-Level Statement

For existing functions and programs, the system skips to the next top-level statement in some circumstances, such as a field not found error. This never happens in an application class program. The code is always executed or an error is produced.

Passing Parameters in Application Class Methods

The parameters to a method are passed by value in the absence of an *out*specification in the parameter list. However, if a parameter list has an out specification, that argument (when used in the call of a method) must be a value that can be assigned something, and is passed by *reference*. For example:

```
/* argument passed by reference */
method increment(&value as number out);


/* argument passed by value */
method increment(&value as number);
```

This is in contrast with existing PeopleCode functions that pass *all* parameters by reference when they can be assigned something. This means the method signature is a specification of which parameters can be updated by the method. This makes the code easier to debug and maintain because it's easy to see what can be updated and what can't.

The following is an example of how to do a simple incrementaion of a variable by one.

```
local Number &Mine;

&Mine = 1;

&obj.Increment(&Mine);
```

&Mine now has the value 2.

Passing Parameters with Object Data Types

Parameters with object data types are always passed by reference:

```
/* argument passed by reference  */
method storeInfo(&f as File);
```

If you specify the out modifier for a method parameter with an object data type, it becomes a *reference parameter*. This means that the parameter variable is passed by reference instead of the object that it is pointing at when passed.

For example, if you pass an object parameter with the out modifier:

```
method myMethod(&arg as MyObjectClass);

Local MyObjectClass &o1 = create MyObjectClass("A");
Local MyOtherObjectClass &o2 = create MyOtherObjectClass();

&o2.myMethod(&o1);
```

And inside myMethod this occurs:

```
Method myMethod
   &arg  =  create MyObjectClass("B");
end-method;
```

Since the method argument is reassigned within the body of myMethod, &o1 does not point at the new instance of MyObjectClass (initialized with "B") after the method call completes. This is because &o1 still references the original instance of MyObjectClass.

However, if &o1 had been passed with the **out** modifier, after the method call completes, &o1 points at whatever the parameter was last assigned to; in this case, the new instance of MyObjectClass. The parameter, rather than the object, is passed by reference.

Using the Out Specification for a Parameter

In the following example, a class, AddStuff, has a single public method, DoAdd. This adds two numbers together, then assigns them as different numbers. In the signature of the method declaration, the first parameter is *not* declared with an out statement, while the second one is.

```
class AddStuff
   method DoAdd(&P1 as number, &P2 as number out);
end-class;

method DoAdd
   &X = &P1 + &P2;
   &P1 = 1;
   &P2 = 2;
end-method;
```

In the following PeopleCode example, an object named &Aref is instantiated from the class AddStuff. Two parameters, &I and &J are also defined.

```
local AddStuff &Aref = Create AddStuff();
local number &I = 10;
local number &J = 20;
```

The following code example is correct. &J is changed, because of the *out*statement in the method signature, and because the value is being passed by reference. The value of &I is *not* updated.

```
&Aref.DoAdd(&I, &J); /* changes &J but not &I */
```

The following code example causes a design time error. The second parameter must be passed *by reference*, not by value.

```
&Aref.DoAdd(10, 20); /* error – second argument not variable */
```

See Also

PeopleCode Developer's Guide, PeopleCode Language

Constructors

The *constructor* for a class is the public method with the same name as the (short name of the) class. The statements contained in this method (if any) provide the initialization of the class.

This constructor is *always* executed when an object of the class is instantiated.

Note. Before a constructor runs, the instance variables for the class are set by default based on their declared types.

Instantiate new objects of an application class by using the **Create** function. This function takes the name of the class, and any parameters needed for the constructor method.

If the short class name is unambiguous, that is, only one class by that short name has been imported, you can use just the short class name.

The following code instantiates an object of the class Fruit:

```
&Fref = Create Fruit();
```

If there's a possibility that the name is ambiguous, you can use the full class name to instantiate the object.

The following code instantiates an object of the Invoice class:

```
&InvObj = Create pt:examples:Invoice();
```

If the Create function is used in the context of further object expressions, that is, with a reference to members by a dot operation, the function call must be enclosed in parentheses. This is to emphasize that the creation happens before the member reference. For example:

```
&InvCust = (Create Invoice()).Cust();
```

If necessary, the constructor is supplied its parameters by the call-list after the class name in the create statement. For example:

```
&Example = create Example(&ConstructorParameter1, 2, "Third");
```

A class that does not extend some other class does not need any constructor.

A class that extends another class must have a constructor, and in the constructor, it must initialize its superclass.

To initialize a superobject, an instance of the superclass is assigned to the keyword **%Super** in the constructor for the subclass. This assignment is allowed only in the constructor for the subclass.

The following example shows this type of assignment:

```
class Example extends ExampleBase
   method Example();
   ...
end-class;

Global string &CurrentBaseString;

method Example
   %Super = create ExampleBase();
   &BaseString = &CurrentBaseString;
   &SlashString = &BaseString;
   &ImportantDate = Date(19970322);
end-method
```

Note. Application classes don't have destructors, that is, a method called just before an object is destroyed. The PeopleCode runtime environment generally cleans up any held resources, so they're not as necessary as they are in other languages.

See Also

Chapter 6, Application Classes , Superclass Reference , page 619

External Function Declarations

External function declarations are allowed in application classes, in the global and component variable declarations, after the class declaration (after the **end-class** statement), and before the method definitions. For example:

```
import SP:Record;

class Person extends SP:Record
   method Person(&pid As string);

   method getPerson() Returns Record;
   method getUserName() Returns string;

private
   instance Record &recPerson;

end-class;

Declare Function getUserName PeopleCode FUNCLIB_PP.STRING_FUNCTIONS FieldFormula;

method Person
   /+ &pid as String +/

   %Super = create SP:Record(Record.SPB_PERSON_TBL);
   &recPerson = %Super.getRecord();
   &recPerson.PERSON_ID.Value = &pid;

end-method;

method getPerson
   /+ Returns Record +/

   Return &recPerson;

end-method;

method getUserName
   /+ Returns String +/
   Local string &formattedName;

   &formattedName = getUserName(&recPerson.PERSON_ID);
   Return &formattedName;

end-method;
```

When application class properties and instance variables are used as the argument to functions or methods, they are always passed by value, never by reference. This means that the called function or method cannot change the value of the property or instance variable. If you want a function call to change a property or instance variable, use a local variable in the call and then assign the property or instance variable after the call.

Suppose you have the following class and you want to call Meth2 in the body of Meth1, passing it MyProp:

```
class MyClass
   property number MyProp;
   method Meth1();
   method Meth2(&i as number out);
end-class;
```

The following PeopleCode *fails* with a compile-time error:

```
method Meth1
   Meth2(%This.MyProp);   /* error - field or temp name required. */
   Meth2(&MyProp);   /* error - field or temp name required. */
end-method;
```

The following PeopleCode is valid:

```
method Meth1
/* Assignment needed if &i is input as well as output in Meth2. */
   local number &Var = %This.MyProp;
   Meth2(&Var);
   %This.MyProp = &Var;
end-method;
```

Note. You call a Java program from an Application Class the same way you do using any other PeopleCode program, that is, by using one of the existing Java class built-in functions.

See Chapter 16, Java Class, From Java to PeopleCode, page 964.

Naming Standards

This section describes the naming standards for:

- Packages.
- Classes.
- Methods.
- Properties.

Naming Packages

The naming standard for application packages is in the following format.

```
CC_XXX[_YYY]
```

Where CC is the company name (for example, PS is PeopleSoft), XXX is the product line, and YYY is the product code.

Examples are PS_CRM and PS_CRM_RB.

Naming Classes

Use a constructor to be the same name as the short name for the class. PeopleSoft recommends that you do not name classes to be GetXxx.

Naming Methods

Make method names simple verb or verb object words, spelled out, first letter of each word capitalized. Examples: GetField, Save, SetDefault, WriteRecord

Naming Properties

Make property names nouns, likewise spelled out, first letter of each word capitalized. Examples: Dimension, HTTPMethod, PathInfo, SubscriptionProcessId.

For Boolean read-only properties, sometimes IsXxx is appropriate. Examples: IsOpen, IsNewFieldID, IsChanged, IsDeleted

State Boolean properties positively, that is, IsChanged, as opposed to negatively, IsUnChanged. This avoids double negatives when testing (If Not &MyField.IsUnChanged Then. . . .)

Import Declarations

Before you can use a class name in a PeopleCode program (either a class definition program or an ordinary PeopleCode program), you must first import it using an import statement at the beginning of your program.

An import statement names either all the classes in a package or one particular application class.

If you're importing all the classes in a package, the syntax for the import statement ends with the following:

```
:*
```

This makes all the application classes directly contained in the named package available. Application classes contained in subpackages of the named package are *not* made available.

If you're importing one particular application class, the syntax of the import statement ends with the short name of the class. Only that class is made available.

For example, the following imports all the classes of the package named Fruit:

```
Import Fruit:*;
```

The following imports only the application classes Banana and Kiwi. If the package named Fruit contains other classes, those are not made available to the PeopleCode program:

```
Import Fruit:Banana;
```

```
Import Fruit:Kiwi;
```

If you had a subpackage in Fruit named Drinks, you could access all the classes in that subpackage like this:

```
Import Fruit:Drinks:*;
```

A PeopleCode program (either a separate program or one contained within a class definition) can use only application class names that have been imported. However, a class definition can use its own name without having to import it.

It's possible to import two packages which contain application classes with the same short name. For example, suppose my package Fruit contained classes Apple and Banana. Suppose a second package, Drinks, contained Apple and Cherry.

```
Import Fruit:*;

Import Drinks:*;
```

You could refer to the Banana and Cherry classes using just the short name of the class. However, you couldn't refer to the Apple class by the short name. You must always use the full name reference (that is, the name of the package and the name of the class) when referring to Apple.

```
Global Banana &MyBanana;

Local Cherry &CherryTree;

Component Fruit:Apple &GreenApple;
```

The following line produces an error, because the program doesn't know which Apple class you're referring to:

```
Component Apple &GreenApple;
```

Note. Single class imports (such as, Import Fruit:Banana) are checked for existence when you save your PeopleCode.

Self-Reference

A method can refer to the current object using the **%This** system variable.

Due to subtyping, %This is an object of either the method's class or a subclass of the method's class.

In PeopleCode, the methods and properties of the class *cannot* be used without an object qualification.

In the following code example, the line in bold indicates an error. You can't call the class this way: you must use %This.

```
class FactorialClass
   method factorial(&I as number) returns number;
end-class;

method factorial
   if &I <= 1 then
      return 1;
   end-if;
```

```
      return &I * factorial(&I – 1); /* error – factorial undefined */

      return &I * %This.factorial(&I – 1); /* okay */
   end-method;
```

One of the implications of this is that when you're writing a method, there isn't any way to guarantee you'll access your own class' version of a public method (rather than a subclass') as it might be overridden by a subclass. The only way to make sure that you can access your own method is to make it private.

If you have some action that you also want to make public, and you want guaranteed access to it, you can make the action a private method, then define a public method for it that calls the private one.

In the following example, factorial is the public method that can be used by other classes. myFactorial is the private action.

```
class FactorialClass
   method factorial(&I as number) returns number;
private
   method myFactorial(&I as number) returns number;
end-class;

method factorial
   return myFactorial(&I);
end-method;

method myFactorial
   if &I <= 1 then
      return 1;
   end-if;

   return &I * %This.myFactorial(&I – 1);

end-method;
```

If you declare an instance variable as private you can still access it as a private property in another instance of the same class. For example, given the following declaration:

```
class Example

  private

     instance number &Num;

end-class;
```

A method of Example could reference another Example instance's &Num instance variable as follows:

```
&X = &SomeOtherExample.Num;
```

Properties and Constants

This section discusses the following ways to use properties and constants with application classes:

- Differentiating between properties and methods.
- Overriding properties.
- Using the Get and Set keywords.
- Using read-only properties.
- Using methods and properties for collections.
- Declaring constants.

Differentiating Between Properties and Methods

A *method* is a procedure or routine, associated with one or more classes, that acts on an object.

A *property* is an attribute of an object.

PeopleSoft recommends that you use read-write or read-only properties instead of creating methods named GetXxx and SetXxx.

For example, instead of creating a method GetName, that takes no parameter, just to return the name of an object, create a read-only property Name.

Application classes can have read-only properties, but no indexed properties.

Considerations Creating Public Properties

There are two implementations of (public) properties:

- as instance variables.
- as get-set methods.

From outside the class, you cannot tell the difference between them (for example, both are dynamically bound). However, if you just want to expose an instance variable to users of the class, you can declare a property of that name and type, which declares the instance variable too:

```
class xxx
     property string StringProp;
end-class
```

If you want the property to be read-only outside the class, use the following code:

```
class xxx
     property string StringProp readonly;
end-class
```

Both of these code examples declare an instance variable called `&StringProp` and provide its value for getting and change its value for setting (if not readonly). You can think of this as shorthand for the usual implementation of get-set methods.

If you want to implement the same with code (perhaps deriving the property from other instance variables or other data), use the following sort of PeopleCode program:

```
class xxx
   property string StringProp get set;
end-class;

get StringProp
   return "string"; /* Get the value from somewhere. */
end-get;

set StringProp
   /* Do something with &NewValue. */
end-set;
```

To specify a read-only property, omit the set portions of the code example.

Overriding Properties

A property of a superclass can be overridden by a subclass by providing a different implementation.

The properties of a class are implemented by the keywords **Get** and **Set**, or by access to an otherwise private instance variable.

To access the property, you do not have to specify Get or Set: the context tells the processor which task you're doing. To create read-only properties, specify only a Get method, with no Set method, or for instance variable properties, specify Readonly.

For an instance variable property, there are two different ways for method code in the class itself to refer to the property:

- %This.*PropertyName*
- &*PropertyName*

The first way is dynamically bound, as it uses whatever (possibly overridden by a subclass) implementation of *PropertyName* that the current object provides.

The second way always accesses the private instance variable of the class that the method is declared in.

The following example returns strings, by converting the property. It also demonstrates overriding.

```
class A
   property number Anum;
   property string Astring get;
   property string Astring2 get;
end-class;

/* The following is a dynamic reference to the Anum property of the current⇒
 object, which might be a subclass of A, so this might not access the A class'⇒
 implementation of Anum as an instance variable. */
   Get Astring
   return String(%This.Anum);
end-get;
```

```
/* The following is a static reference to our &Anum instance variable. Since it⇒
 does not use %This, it will not be overridden even if the current object is a⇒
 subclass of A. */
Get Astring2
   return String(&Anum);
end-get;

/* The following overrides the Anum property of our superclass,
in this case by providing another instance variable implementation.
This gives us our own &Anum instance variable. */
class B extends A
   property number Anum;
end-class;
. . .
/* Set the Anum property in B. Because of the above definitions, this sets
the &Anum instance variable in the B object, but not the &Anum instance
variable in its superobject. In the absence of other setting, the latter will
be defaulted to 0. */

local B &B = Create B();
&B.Anum = 2;

&MyValue = &B.Astring; /* &MyValue is "2" */
&MyValue = &B.Astring2; /* &MyValue is now "0" */
```

Using Get and Set with Properties

PeopleSoft recommends against using read-only properties that are public and mutable in application classes (similar to public instance variables) because this allows external code to read and change the internal state of the object without any controls. Instead, modify the property declarations with the **Get** and **Set** keywords.

There is a slight performance advantage, and you write less code, if you use the **Get** and **Set** keywords over using get and set methods.

If you define a property with the**Get** and **Set** keywords, you also must define a get or set block in the same part of the code that contains the method bodies.

Using Read-Only Properties

PeopleCode read-only properties are functionally similar to static variables, as long as the class that defines the properties initializes them and does not change them. The difference is that you need an instance of the class to access them.

For example, consider a Category object that has a CategoryType member that can have one of the following values:

- ‘R’ for Regular
- ‘B’ for Browse
- ‘N’ for News

You can define three read-only properties corresponding to each of these values, REGULAR, BROWSE, and NEWS, then initialize them to the appropriate values in the constructor of the class.

When you write code to evaluate the CategoryType of the Category object, perform a test like this:

```
If (&category.getCategoryType() = &constants.NEWS) then...
```

PeopleSoft recommends referring to the read-only properties, rather than their literal values, for the following reasons:

- If you spell a literal value incorrectly, the PeopleCode editor does not catch it; if you spell a read-only property wrong, it does.
- If you ever need to change the value of a read-only property, you only have to do it in one place.
- It makes the code more maintainable and reusable (because NEWS is more intelligible than ‘N’).

Using Methods and Properties for Collections

Application classes can contain collections. Collections of objects generally have at least the following methods and properties.

- First()
- Item(*Index*)
- Next()
- Count (as a property)

All numeric indexes in PeopleCode are one-origin, that is, start all indexes at 1, not 0.

Declaring Constants

PeopleSoft recommends that if constants are inextricably linked to a class, you should define them within that class. If the constants are more general, potentially used by multiple classes, consolidate the constants into a constants class at application-level or package-level.

Message catalog set and message numbers are good candidates to centralize into a single constants class. It can improve the readability of code and make it easier to maintain these values.

For example, you can define a constant MSG_SET_NBR, and then define constant values for each message that explain a little about what the message represents (like REQUIRED_VALUE_MISSING, UNIMPLEMENTED_METHOD, or INSUFFICIENT_PRIVILEGES).

Using Variables in Application Classes

This section discusses how to use the following variables in application classes:

- Instance variables.
- Global variables.
- Overriding variables and properties.

Declaring Private Instance Variables

A private instance variable is private to the class, not just to the object instance. For example, consider a linked-list class where one instance needs to update the pointer in another instance. Another example is the following class declaration:

```
class Example private instance number &Num; end-class;
```

A method of Example could reference another instance of the Example &Num instance variable as:

```
&X = &SomeOtherExample.Num;
```

Avoid making every instance-scoped variable a public property. You should consider each variable individually and decide if it belongs in the public interface or not. If it does, decide if the variable warrants get or set modifiers and therefore should be a public property. If the variable only serves internal purposes to the class, it should be declared as a private instance variable.

Global Variables

PeopleSoft recommends avoiding global variables in application classes unless there is a situation where no other code works. Generally, an application class should not know about anything scoped outside of its instance. There are exceptions, since not all application classes should be standalone. For example, an application class that interacts with the Component Buffer must know about the Component Buffer objects.

Overriding Variables and Properties

PeopleSoft recommends against overriding of variables and of properties.

Since instance variables are private in scope, you cannot access them in subclasses, but you can redeclare them in subclasses. The same is true of public properties; you can even explicitly access the overridden or overriding property. However, overriding is not recommended.

Superclass Reference

A method can refer to a member of its superclass by using the **%Super** system variable. This construction is needed only to access superclass members that are hidden by overriding members in the current class.

In the following example, the classes that extend the general BuildingAsset class each have their own method DisasterPrep, that does the specific processing, then calls the superclass (generic) version of DisasterPrep.

```
/* generic building class */
class BuildingAsset
   method Acquire();
   method DisasterPrep();
end-class;

method Acquire
   %This.DisasterPrep();
end-method;
```

```
method DisasterPrep
   PrepareForFire();
end-method;

/* building in Vancouver */
class VancouverBuilding extends BuildingAsset
   method DisasterPrep();
end-class;

method DisasterPrep
   PrepareForEarthquake();
   %Super.DisasterPrep(); /* call superclass method */
end-method;

/* building in Edmonton */
class EdmontonBuilding extends BuildingAsset
   method DisasterPrep();
end-class;

method DisasterPrep
   PrepareForFreezing();
   %Super.DisasterPrep(); /* call superclass method */
end-method;

local BuildingAsset &Building = Create VancouverBuilding();

&Building.Acquire(); /* calls PrepareForEarthquake then PrepareForFire */

&Building = Create EdmontonBuilding();

&Building.Acquire(); /* calls PrepareForFreezing then PrepareForFire */
```

Downcasting

Downcasting is the process of determining if an object is of a particular subclass type. If the object has that subtype (either the object is of that subtype, or is a subtype of it), a reference to the subobject is returned, otherwise Null is returned. In either case, the type of the resulting value is compatible with the named subclass type.

This downcast inquiry into the actual type of an object can be used when the object has been passed to some general facility, then passed back to more specific code. For example, the general facility might be a dispatch mechanism, or a new storage structure such as a binary balanced tree. These general facilities won't know about specific types, but it might be necessary for the specific program using the general facility to recover the actual type of an object that is returned to it by the general facility.

A downcast should *not* be used when you could add members to a common base class to provide the same functionality. In other words, if you find yourself using the downcast to test whether the object is one or another of several related types, you should add members to the common base class to provide the operations or properties that you are trying to access, then provide separate implementations of those actions or properties in each specific class.

If the downcast operator is to be used in the context of further object expressions (such as references to other members using dot notation) then the downcast must be enclosed in parentheses to emphasize that the downcast happens before the member reference.

```
class Fruit
   property number FruitCount;
end-class;

class Banana extends Fruit
   property number BananaCount;
end-class;
```

In the following code example, first the variables are assigned. All the variable assignments are legal, for Banana is a subtype of Fruit.

```
local Banana &MyBanana = Create Banana();
local Fruit &MyFruit = &MyBanana;  /* okay, Banana is a subtype of Fruit */
local number &Num = &MyBanana.BananaCount;
```

The next two lines of code don't produce an error, as &MyFruit is currently a Banana at runtime.

```
&MyBanana = &MyFruit as Banana;

&Num = (&MyFruit as Banana).BananaCount; /* same as &MyBanana.BananaCount */
```

In the next lines of code, we're reassigning &MyFruit to be of type Fruit, that is, of the superclass Fruit. When you try to reassign &MyFruit as Banana, &MyBanana is assigned as Null, because &MyFruit is no longer of type Banana.

```
&MyFruit = Create Fruit();

&MyBanana = &MyFruit as Banana; /* assigns Null - &MyFruit isn't a Banana */
```

Exception Handling

Use the Exception Class to do exception handling in your PeopleCode. This class provides a try-catch and throw mechanism so you don't need to check after each operation for errors, but can instead declare by the structure of the try-catch statements when you are interested in handling exceptions, and how you want to handle them.

PeopleSoft recommends that when applicable, application class methods should throw exceptions instead of communicating back to the calling code with return values. Exceptions are instances or subclasses of the PeopleCode Exception class. You can create exceptions by:

- Creating an Exception base class that encapsulates the built-in function call and handles its function parameters consistently. This is the preferred way.

- Calling the built-in function **CreateException**.

Since a method can throw exceptions for several different reasons, PeopleSoft recommends that you:

- create subclasses of Exception often.
- throw strongly-typed exceptions.

This allows the calling code to make decisions based on the exception type.

For example, a method that retrieves customer data from a database might throw an exception named SQLException if a database error occurs, or an exception named PrivilegeException if the current user doesn't have the appropriate permissions to perform the operation. There could be different errors causing either of these exception types to be thrown in this method; if necessary, you can write the calling code to examine the specific message numbers and react accordingly.

PeopleSoft recommends that you always catch an exception; if it is not caught, it causes a runtime error. Use a try-catch block.

Not catching an exception causes a runtime error and can cause PeopleTools to display a dialog box with the message catalog entry and information about the technical context of the exception. The reasons for avoiding this by using try-catch blocks are:

- Application code often continues some form of processing even in the event of an exception; you should not assume that the script stops all processing if an exception occurs.
- The error message that PeopleTools displays might not be appropriate for end-users because of its technical information.

See Also

Chapter 11, Exception Class , page 835

Chapter 1, PeopleCode Built-in Functions, CreateException , page 97

Chapter 1, PeopleCode Built-in Functions, Throw , page 457

Chapter 1, PeopleCode Built-in Functions, Try , page 482

Commenting and Documenting Application Classes

This section contains guidelines for writing method header comments, class header comments, and tags inside the comments. It discusses:

- Comments and documentation
- Method header comments
- Class header comments

Understanding Comments and Documentation

Documentation is important in all code, but even more so in application class code, which aims at, and incurs certain overhead in the name of, higher levels of reuse.

There are two types of inline comments:

- Internal comments, which start with /* and end with */.
- Comments which can potentially be used to generate API documentation, which start with /** and end with */. You can write method header and class header comments for application classes, and you can use the comment tags, with this type of comment.

PeopleSoft recommends keeping a running line of asterisks along one side of an extensive comment to help the readability of the code by making the entire comment stand out from the surrounding code.

Method Header Comments

The following is an example of documentation conventions for method header comments, which are also suitable for PeopleCode. It contains comment tags, which are explained below the example.

```
/**
  * Description of what method does
  *
  * @param a optional description of param a
  * @param b optional descriptions of param b
  * @exception PrivilegeException thrown if etc, etc
  * @exception SQLException thrown if etc., etc.
  * @return String  -  description of potential values
  */
Method myMethod
   ...
End-method;
```

Method Comment Tags

The following are the suggested tags for commenting methods.

@param *N*	Specify a description for each method parameter. The name of the parameter is specified after the keyword @param.
@exception *name*	Specify a description of each exception class. The name of the exception is specified after the keyword @exception. For example, the descriptions can explain what causes certain exceptions to be thrown.
@return *Type*	Specify a description of the return value. The type of data that is returned is specified after the keyword @return. For example, explain the potential values a method may return, whether it returns null in some circumstances, and so on.

Not all methods require documentation. For example, some simple set or get methods have method signatures that are descriptive enough. In other cases, it may suffice to include a brief sentence in the header explaining what the method does.

As with methods, PeopleCode get or set blocks should also carry method header comments.

Class Header Comments

Class header comments should provide descriptive information about the class, and may optionally include additional tags, indicating version and author:

```
/**
* class description
*
* @version 1.0
* @author amachado
*/
```

Class Header Comment Tags

The following are the suggested tags for commenting class headers.

@version *number*	Specify a version number for a class.
@author *name*	Specify an author for a class.

Designing Base Classes

This section discusses some ways you can design application classes.

- Base data classes
- Abstract base classes
- Generic base classes
- Utility classes

Base Classes

Base classes contain a representation of data elements, such as documents, categories, users, and miscellaneous security-related entities.

To design the document data element to be integration-friendly, PeopleSoft recommends you start with a relatively basic base class. Then use more specialized types of documents to create a subclass from that base class, adding the necessary specialized attributes.

Even if you think of ways to use data to drive the business rules, PeopleSoft recommends that you do not place them into your data objects. Replicating business rules for every instance of a data element can be detrimental to performance.

Abstract Base Classes

Abstract base classes contain reusable logic for PeopleCode classes; you do not instantiate them directly. Instead, you instantiate classes that are subclasses of the abstract base class. Abstract base classes use placeholder methods for specialized logic, and its subclasses override those methods to provide the specialized logic.

An example of a candidate for an abstract base class is a class that is used for administering a data entity, such as an invoice. Such a class has a Save method. A Save method can use normal methods to call generic validation routines, talk to the database, and handle errors; it can use the placeholder methods for specialized logic, such as mapping a given data class to a particular database table.

PeopleSoft recommends that you program an active validation check into your placeholder method that causes a runtime error, such as throwing an UnimplementedMethodException, if an abstract base class placeholder method hasn't been overridden.

Generic Base Classes

Generic base classes contain reusable logic for PeopleCode classes; you write them to be self-contained, fully implemented, and instanitable. This differs from abstract base classes, which are not instantiated directly and need subclasses that override methods to provide specialized logic.

Some examples of classes that are good candidates for creating generic base classes are:

- Service classes, which typically extend a class that provides infrastructure services, such as configurable application logging.
- Data classes, which typically extend a class that enables you to plug in adapters to support sorting their instances with custom sort algorithms.

Even when a class does not seem generic because it only has a couple of subclasses, you can still define it in your design as a generic base class rather than cloning common code across different classes. The main reason is maintainability: it's easier to maintain a single class than trying to keep track of the same code scattered across your database.

Utility Classes

Some reusable code is the kind you typically don't want to inherit, but you just want to use here and there. Utility classes can serve this purpose.

String parsing, data type conversion, and formatting routines are all good candidates for utility classes. PeopleSoft recommends that anytime you code something that's more than just a line or two, and that seems to be reusable, write it as a utility class.

Declaration of Application Classes

This section discusses instructions for declaring application classes, such as:

- Declaring application classes.
- Importing class names.
- Referencing Superclasses.
- Declaring private methods.

Declaring Application Classes

Application object references are declared in your PeopleCode programs by either the short name of the class (if the short name is unambiguous amongst all the imported classes), or by the full name of the class. Like all uses of class names, to use the name in a variable declaration, you must have imported the class. Application objects can be of Local, Global, or Component scope.

Every application class is a definition of a new data type in PeopleCode, like the record, rowset, or field data types. The application class includes both the definition of its external interface, its data (properties and instance variables), and its actions (methods), just like the PeopleSoft delivered classes.

Importing Class Names

When you import a class name, fully-qualifying it is optional. The syntax is:

```
<package>:<subpackage>:<...>:<classname or wildcard>
```

PeopleSoft recommends that you individually import each class when you import multiple classes from the same package, instead of using wildcards.

Referencing Superclasses

PeopleSoft recommends that you do not set **%Super** to a new instance of the current class you are defining because this would start an infinite loop.

```
/* Do not do this! */
%Super = %This;
```

Declaring Private Methods

PeopleSoft recommends that you use private methods for succinct operations, even if they aren't reused, to improve the readability and maintainability of the code. You can also do this by using many inline comments.

Scope of an Application Class

Application classes can be instantiated only from PeopleCode. These classes can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

Application Classes Built-in Functions and Language Constructs

In this section, we discuss the Application classes built-in functions and language constructs, in alphabetical order.

Class

Syntax

```
Class classname [Extends Classname]
  [Method_declarations]
   [Property_declarations]
[Private
   [Method_declarations]
   [Instance_declarations]
   [Constant_declaration]]
End-Class
```

Where Method_declarations are of the form:

```
Method methodname([MethodParam1 [, MethodParam2. . .] [Returns Datatype]])
```

Where Property_declarations are of the form:

```
Property DataType PropertyName {{[Get] | [Set]} | [readonly]}
```

Where Instance_declarations are of the form:

```
Instance DataType &Variable1 [, &Variable2. . .]
```

Where Constant_declarations are of the form:

```
Constant &Constant = {Number  | String | True | False | Null }
```

Description

Use the **Class** language construct to build application classes in an application package. All classes within a package must be uniquely named, however, classes contained in different packages do not have to be named uniquely.

External function declarations are allowed in application classes, in the global and component variable declarations, after the class declaration (after the **end-class** statement), and before the method definitions.

Parameters

Parameter	Description
classname	Specify the name of the class that you are creating. All classes within a package must be uniquely named, however, classes contained in different packages do not have to be named uniquely.
Extends *classname*	Specify the name of the class that this class extends. You must import the class to extend it.
Method_declarations	Specify the methods (and their signature) as used in this class.
Property_declarations	Specify the properties, and their usage, in this class.
Private	Use this keyword to declare any methods, constants, or instance variables as private to the class, that is, they can't be accessed by any other classes.
Instance_declarations	Specify any variables that should be in any instance (object) of the class.
Constant_declarations	Specify any constants that should be used with this class.

Returns

None.

Example

```
class Example extends ExampleBase
   method NumToStr(&Num as number) returns string;
   method AppendSlash();
   property number SlashCount get;
   property number ImportantDayOfWeek get set;
   property string SlashString readonly;
   property date ImportantDate;
private
   method NextDayOfWeek(&DoW as number) returns date;
   constant &Sunday = 1;
   instance number &BaseString;
end-class;
```

See Also

Chapter 6, Application Classes , Method , page 629, Chapter 6, Application Classes , Get , page 628, Chapter 6, Application Classes , Set , page 630.

Get

Syntax

```
Get PropertyName
StatementList
End-Get
```

Description

Use the **Get** language construct when defining properties in an application class that are implemented by methods rather than an instance variable. All properties within an application class must be uniquely named.

Parameters

Parameter	Description
Get *PropertyName*	Specify the name of the property that you're implementing.
StatementList	Returns the value of the property. In other words, this is a method which has no parameters and must return the value of the property.

Returns

Depends on the assignment within *StatementList*

Example

```
Get FruitCount

  Return &MyFruit.Number();
```

```
End-Get;
```

See Also

Chapter 6, Application Classes , Method , page 629, Chapter 6, Application Classes , Set , page 630, Chapter 1, PeopleCode Built-in Functions, Return , page 376.

Import

Syntax

Import *PackageName*:[*PackageName. . .*:]{*Classname* | *****}

Description

Before you can use a class name, you must first import it using an import declaration statement at the beginning of your program.

An import declaration statement names either all the classes in a package or a single application class.

Parameters

Parameter	Description
Import *PackageName*	Specify the name of the package that you want to import.
*****	Import all the classes of the specified package.
Classname	Import only this class of the specified package. You won't have access to any other classes in the specified package, only this one.

Returns

None.

Example

The following gives you access to all the classes in the package Fruit:

```
Import Fruit:*;
```

The following only gives you access to the class Banana in the package Fruit:

```
Import Fruit:Banana;
```

The following gives you access to the class Apple, in the package Drinks, which contains the package Fruit:

```
Import Drinks:Fruit:Apple;
```

See Also

Chapter 6, Application Classes , Class , page 627.

Method

Syntax

Method MethodName

```
StatementList
End-Method
```

Description

Use the **Method** statement to define the methods of your application class.

Parameters

Parameter	Description
Method *methodname*	Specify the name of the method that you are creating.
StatementList	Specify the program of the method, what it does.

Returns

Depends on the method.

Example

```
method NumToStr
   return String(&Num);
end-method;

method AppendSlash
   &SlashString = &SlashString | "/";
end-method;
```

See Also

Chapter 6, Application Classes , Get , page 628, Chapter 6, Application Classes , Set , page 630, Chapter 1, PeopleCode Built-in Functions, Return , page 376.

Set

Syntax

```
Set PropertyName
StatementList
End-Set
```

Description

Use the **Set** language construct when implementing properties in an application class. All properties within an application class must be uniquely named. The new value for the property is available in the &NewValue parameter of the Set method.

Note. You cannot create a set-only property. You can create only get-set properties.

Parameters

Parameter	Description
Set *PropertyName*	Specify the name of the property that you're implementing.
StatementList	Change the value of the property to be &NewValue.

Returns

None.

Example

```
Set FruitCount

  &Fruitcount = &MyFruit[&NewValue].ActiveRowCount();

End-Set;
```

See Also

Chapter 6, Application Classes , Get , page 628, Chapter 6, Application Classes , Method , page 629.

CHAPTER 7

Array Class

This chapter provides an overview of arrays and discusses how to:

- Create arrays.
- Populate an array.
- Remove items from an array.
- Create empty arrays.
- Create and populate multi-dimensional arrays.
- Use flattening and promotion.
- Declare array objects.
- Understand the scope of an array object.
- Use array class built-in functions.
- Use array class methods
- Use array class properties

Understanding Arrays

An array is a collection of data storage locations, each of which holds the same type of data. Each storage location is called an element of the array. When you create an array, you don't have to declare the size of the array at declaration time. Arrays grow and shrink dynamically as you add (or remove) data. The size of an array is limited by the available memory. You can't access past the end of an array, but you can assign outside the existing boundaries, thereby growing the array.

Creating Arrays

Arrays are declared by using the Array type name, optionally followed by "of" and the type of the elements. If the element type is omitted, it is set by default to ANY.

```
Local Array of Number &MYARRAY;
Local Array &ARRAYANY;
```

Arrays can be composed of any valid PeopleCode data type, such as string, record, number, date, and so on.

PeopleSoft recommends you declare every object you use in PeopleCode. This provides some syntax checking when you save PeopleCode. It's better to find out that you misspelled the name of a method or property at design time, rather than at runtime!

Arrays can be declared as Local, Global, or Component, just like any other PeopleTools object.

Arrays can be created with one or more dimensions. An array with more than one dimension is called an array of arrays. The *dimension* of an array is the number of Array type names in the declaration. This is also called the *depth* of an array. The maximum depth of a PeopleCode array is 15 dimensions.

In the following example, &MYARRAY has three dimensions, and &MYA2 has two dimensions.

```
Local Array of Array of Array of Number &MYARRAY;
Local Array of Array &MYA2;
```

An array must always have a consistent dimension. This means that in a one-dimensional array *none* of the elements can be an array, in a two-dimensional array *all* of the elements must be one-dimensional arrays, and so on.

After you declare an array, use one of the built-in array functions to instantiate it and return an object reference to it. For example, the following creates an array containing one element of type &TEMP, whatever data type &TEMP may be.

```
&MYARRAY = CreateArray(&TEMP);
```

Or you can use the **CreateArrayRept** function to instantiate an array. The **Rept** stands for *repeat*. CreateArrayRept creates an array that contains the number of copies you specify of a particular value. The following code creates an array with three copies of the string &MYSTRING. This does *not* create a three-dimensional array, but rather creates an array that's already populated with three elements of data (**Len** = 3), each of which contain the same string (&MYSTRING).

```
&MYARRAY = CreateArrayRept(&MYSTRING, 3);
```

An array *object* can be assigned to an array *variable*. Array objects can be passed from and returned to any kind of PeopleCode function:

```
ANewFunc (&myarray);
MyFunc (&myarray);
&MyArray = YourFunc("something");
```

For example, the **ReturnToServer** function returns an array of nodes to which a message can be published.

Elements in an array are specified by providing a bracketed subscript after the array object reference:

```
&MyArray[1] = 123;
&temp = &memory[1][2][3];
&temp = &memory[1, 2, 3]; /* Same as preceding line.  */
MyFunc(&MyArray[7]);
MyFunc(10)[15] = "a string";
```

To access data in a two-dimensional array, you must specify both indexes. The following accesses the second item in the first subarray:

```
&VALUE  = &DOUBLE[1][2];
```

You receive an error if you use a zero or negative index in an array. Accessing an array element whose index is larger than the last array element is also an error, but storing to such an index extends the array. Any intervening elements between the former last element and the new last element are assigned a value based on the element type of the array. This is the same value as an unassigned variable of that type.

An array is an object, which means that assignments to an array are the same as for any other object. An array variable can be assigned the distinguished value NULL, which indicates the absence of any array value.

Array variables are supported for all scopes. This means that you can have local, global,and Component array variables.

See Also

Chapter 1, PeopleCode Built-in Functions, ReturnToServer , page 377

PeopleCode Developer's Guide, Understanding Objects and Classes in PeopleCode, Assigning Objects

Populating an Array

There are several ways to populate an array. The example code following each of these methods creates the exact same array, with the same elements:

- Use **CreateArray** when you initially create the array:

```
Local Array of Number &MyArray;

&MyArray = CreateArray(100, 200, 300);
```

- Assign values to the elements of the array:

```
Local Array of Any &MYARRAY;

&MYARRAY = CreateArray();
&MYARRAY[1] = 100;
&MYARRAY[2] = 200;
&MYARRAY[3] = 300;
```

Note. Using CreateArray without any parameters creates an array of Any.

- Use the **Push** method to add items to the end of the array:

```
Local Array of Number &MYARRAY;
Local Number &MYNUM;

&MYARRAY = CreateArrayRept(&MYNUM, 0);
/* this creates an empty array of number */
&MYARRAY.Push(100);
&MYARRAY.Push(200);
&MYARRAY.Push(300);
```

- Use the **Unshift** method to add items to the beginning of the array:

```
Local Array of Number &MYARRAY;
Local Number &MYNUM;

&MYARRAY = CreateArrayRept(&MYNUM, 0);
/* this creates an empty array of number */
&MYARRAY.Unshift(300);
&MYARRAY.Unshift(200);
&MYARRAY.Unshift(100);
```

You can also use **CreateArrayRept** (repeat)when you initially create the array. The following example creates an array with three elements: all three elements have the same data, that is, 100:

```
Local Array of Number &MYARRAY;

&MYARRAY = CreateArrayRept(100, 3);
```

Removing Items From an Array

You can remove elements from either the start or the end of the array:

- Use the POP method to select and remove an element from the end of an array

```
Local Array of Number &MYARRAY;

&MYARRAY = CreateArray();
&MYARRAY[1] = 100;
&MYARRAY[2] = 200;
&MYARRAY[3] = 300;

&ANSWER = &MYARRAY.Pop();
```

&ANSWER will equal 300.

- Use the SHIFT method to select and remove an element from the beginning of an array

```
Local Array of Number &MYARRAY;

&MYARRAY = CreateArray();
&MYARRAY[1] = 100;
&MYARRAY[2] = 200;
&MYARRAY[3] = 300;

&ANSWER = &MYARRAY.Shift();
```

&ANSWER equals 100.

Creating Empty Arrays

If you create an array using **CreateArray** of any data type other than ANY, the new array is *not* empty: it contains one item. If you need to create a completely empty array that contains 0 elements, use one of the following:

```
Local Array of Number &AN;
Local Array of String &AS;
Local Array of Record &AR;
Local Array of Array of Number &AAN;
Local Record &REC;

&AN = CreateArrayRept(0,0); /* creates an empty array of number */
&AS = CreateArrayRept("", 0); /* creates an empty array of string */
&AR = CreateArrayRept(&REC, 0); /*create empty array of records */
&AAN = CreateArrayRept(&AN, 0); /*creates empty array of array of number */
&BOTH = CreateArray(CreateArrayRept("", 0), CreateArrayRept("", 0)); /* creates an⇒
 empty array of array of string */
```

Creating and Populating Multi-Dimensional Arrays

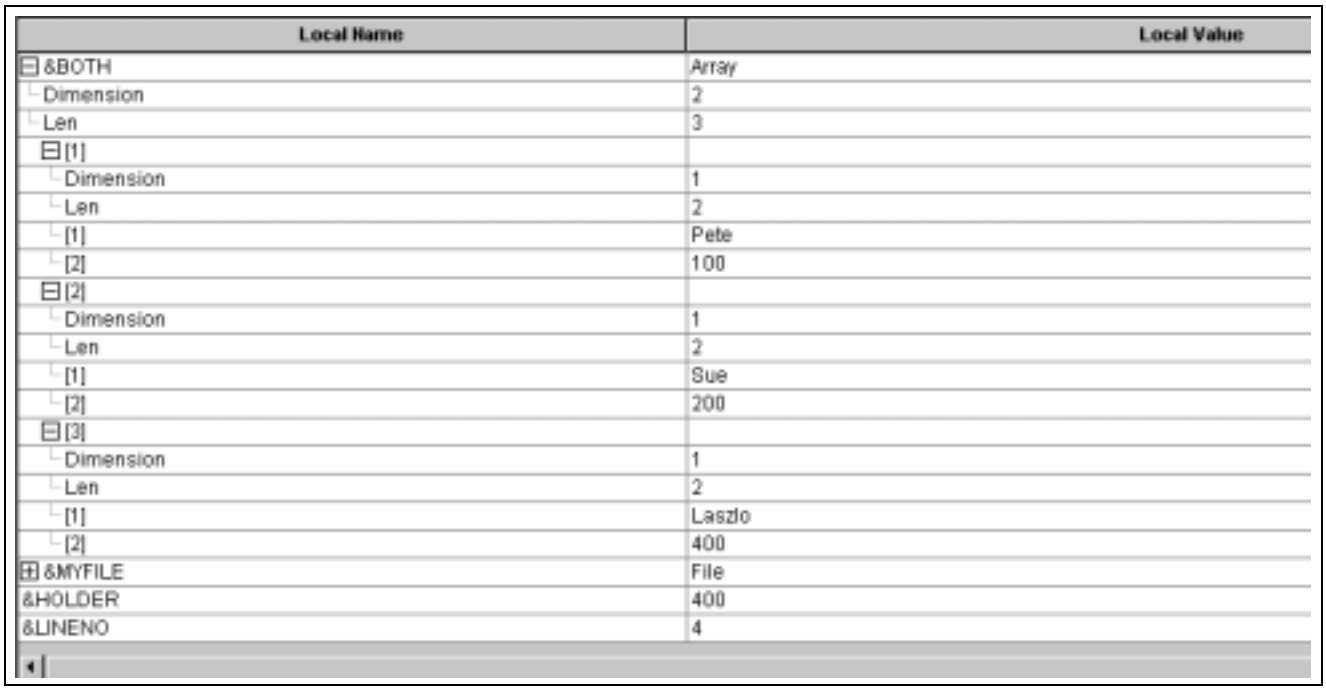
You can create arrays with more than one dimension. Each element in a multi-dimensional array is itself an array. For example, a two-dimensional array is really an array of arrays. Each subarray has to be created and populated as an array.

Each subarray in a multi-dimensional array must be of the same type. For example, you can't create a two dimensional array that has one subarray of type string and a second subarray of type number.

The following example creates an array of array of string, then reads two files, one into each "column" of the array. The **CreateArrayRept** function call creates an empty array of string (that is, the **Len** property is 0) but with two dimensions (that is, with two subarrays, **Dimension** is 2). The first **Push** method adds elements into the first subarray, so at the end of that WHILE loop in the example, &BOTH has **Len** larger than 0. The other **Push** methods add elements to the second subarray.

```
Local array of array of string &BOTH;
Local File &MYFILE;
Local string &HOLDER;

/* Create empty &BOTH array */
&BOTH = CreateArrayRept(CreateArrayRept("", 0), 0);

/* Read first file into first column */

&MYFILE = GetFile("names.txt", "R");
While &MYFILE.ReadLine(&HOLDER);
   &BOTH.Push(&HOLDER);
End-While;

/* read second file into second column */
```

```
&MYFILE = GetFile("numbers.txt", "R");
&LINENO = 1;
While &MYFILE.ReadLine(&HOLDER);
   If &LINENO > &BOTH.Len Then
      /* more number lines than names, use a null name */
      &BOTH.Push(CreateArray("", &HOLDER));
   Else
      &BOTH[&LINENO].Push(&HOLDER);
   End-If;
   &LINENO = &LINENO + 1;
End-While;

/* if more names than numbers, add null numbers */
for &LINENO = &LINENO to &BOTH.Len
      &BOTH[&LINENO].Push("");
End-For;
```



Local Name	Local Value
&BOTH	Array
Dimension	2
Len	3
[1]	
Dimension	1
Len	2
[1]	Pete
[2]	100
[2]	
Dimension	1
Len	2
[1]	Sue
[2]	200
[3]	
Dimension	1
Len	2
[1]	Laszlo
[2]	400
&MYFILE	File
&HOLDER	400
&LINENO	4

&BOTH array expanded in PeopleCode debugger at program end

The following code reads from a two-dimensional array and writes the data from the each subarray into a separate file.

```
Local File &MYFILE1, &MYFILE2;
Local string &STRING1, &STRING2;
Local array of array of string &BOTH;
.
/* code to load data into array would be here */
.
/* open files to be written to */

&MYFILE1 = GetFile("names.txt", "A");
```

```
&MYFILE2 = GetFile("numbers.txt", "A");

/* loop through array and write to files */

For &I = 1 To &BOTH.Len
   &J = 1;
   &STRING1 = &BOTH[&I][&J];
   &MYFILE1.writeline(&STRING1);
   &J = &J + 1;
   &STRING2 = &BOTH[&I][&J];
   &MYFILE2.writeline(&STRING2);
End-For;

&MYFILE1.Close();
&MYFILE2.Close();
```

The following example populates a multi-dimensional string array using SQL. This could be used for reading small tables.

```
Component array of array of string &ArrRunStatus;

&ArrRunStatus = CreateArrayRept(CreateArrayRept("", 0), 0);
&ArrRunStatusDescr = CreateArrayRept("", 0);

&SQL = CreateSQL("SELECT FIELDVALUE, XLATSHORTNAME FROM XLATTABLE WHERE FIELDNAME =⇒
 'RUNSTATUS'");

&LineNo = 1;
While &SQL.Fetch(&FieldValue, &XlatShortName)
      &ArrRunStatus.Push(&FieldValue);
      &ArrRunStatus[&LineNo].Push(&XlatShortName);
      &LineNo = &LineNo + 1;
End-While;
```

To search for a particular element in this array, use the following:

```
&iIndex = &ArrRunStatus.Find(&RunStatusToGet);

&RunStatusDescr = &ArrRunStatus[&iIndex][2];
```

The following example shows how to create a two-dimension array using CreateArrayRept and Push. In addition, it shows how to randomly assigns values to the cells in a two-dimension array.

```
Local array of array of string &ValueArray;

&Dim1 = 10;
&Dim2 = 5;
&ValueArray = CreateArrayRept(CreateArrayRept("", 0), 0);
For &I = 1 To &Dim1
   &ValueArray.Push(CreateArrayRept("", &Dim2));
End-For;
```

```
&ValueArray[1][1] = "V11";
&ValueArray[2][1] = "V21";

WinMessage("&ValueArray[1][1] = " | &ValueArray[1][1] | " " | "&ValueArray[2][1] =⇒
 " | &ValueArray[2][1], 0);
```

Using Flattening and Promotion

Several of the functions and methods that support arrays in PeopleCode use flattening and promotion to convert their operands to the correct dimension for the array.

Flattening converts an array into its elements. For example, the **CreateArray** built-in function constructs an array from its parameters. If it is constructing a one-dimensional array and is given an array as a parameter, then it flattens that array into its elements and adds each of them to the array that it is building, rather than adding a reference to the array (which would be a dimension error) or reporting an error.

Likewise, for functions that operate on multiple-dimension arrays, if they are given a non-array parameter, they use *promotion* to convert it into an array of suitable dimension. For example, the **Push** method appends elements onto the end of an array. If it is operating with a two-dimensional array of Array of Number, and is given a numeric argument, it will convert the argument into a one-dimensional array of Number with the given number as its only element, and then append that to the two-dimensional array.

An array value can only be assigned to an array variable if the value and variable have both the same dimension and base type. This means you cannot assign an Array of Any to an Array of Number variable or vice-versa. You can, however, assign an Array of Number to an Any variable, as long as you do not break the rule that the base element of an array cannot be an array reference value.

Declaring Array Objects

Arrays are declared by using the Array type name, optionally followed by "of" and the type of the elements. If the element type is omitted, it is set by default to ANY.

```
Local Array of Number &MYARRAY;
Local Array &ARRAYANY;
```

Arrays can be composed of any valid PeopleCode data type, such as string, record, number, date, and so on.

Understanding the Scope of an Array Object

An array object can only be instantiated from PeopleCode. This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

Array Class Built-in Functions

Chapter 1, PeopleCode Built-in Functions, CreateArray , page 92

Chapter 1, PeopleCode Built-in Functions, CreateArrayAny , page 93

Chapter 1, PeopleCode Built-in Functions, CreateArrayRept , page 94

Chapter 1, PeopleCode Built-in Functions, Split , page 441

Array Class Methods

In this section, we discuss the array class methods.

Clone

Syntax

Clone()

Description

Clone returns a reference to a new array, which is a copy of the given array. It copies all levels of the array, meaning, if the array contains elements which are themselves arrays (actually references to arrays), then the copy contains references to copies of the subarrays. Furthermore, if the array contains elements that are references to the same subarray, then the copy contains references to *different* subarrays (which of course have the same value).

Parameters

None. The array object that the clone method is executed against is the array to be cloned. Assigning the result of this method assigns a reference to the new array.

Returns

An array object copied from the original.

Example

In the following example, &AAN2 contains the three elements like &AAN, but they are distinct arrays. The last line changes only &AAN2[1][1], *not* &AAN[1][1].

```
Local Array of Array of String &AAN, &AAN2;

&AAN = CreateArray(CreateArray("A", "B"), CreateArray("C", "D"), "E");
&AAN2 = &AAN.Clone();
&AAN2[1][1] = "Z";
```

After the following example, &AAN contains three elements: two references to the subarray that was &AAN[2] (with elements C and D), and a reference to a subarray with element E.

```
&AAN[1] = &AAN[2];
```

After the following example, &AAN2 contains three elements: references to two different subarrays both with elements C and D, and a subarray with element E.

```
&AAN2 = &AAN.Clone();
```

See Also

Chapter 7, Array Class , Subarray, page 653.

Find

Syntax

Find(*value*)

Description

For a one-dimensional array, **Find** returns the lowest index of an element in the array that is equal to the given value. If the value is not found in the array, it returns zero.

For a two-dimensional array, **Find** returns the lowest index of a subarray which has its first element equal to the given value. If such a subarray is not found in the array, it returns zero.

Note. This method works with arrays that have only one or two dimensions. You receive a runtime error if you try to use this method with an array that has more than two dimensions.

Parameters

Parameter	Description
value	The string or subarray to search for.

Returns

An index of an element or zero.

Example

Given an array &AS containing (A, B, C, D, E), the following code sets &IND to the index of D, that is, &IND has the value 4:

```
Local array of string &AS;

&AS = CreateArrayRept("", 0);
&AS.Push("A");
&AS.Push("B");
&AS.Push("C");
&AS.Push("D");
&AS.Push("E");
&IND = &AS.Find("D");
```

Given an array of array of string &AABYNAME containing (("John", "July"), ("Jane", "June"), ("Norm", "November"), the following code sets &IND to the index of the subarray starting with "Jane", that is, &IND has the value 2:

```
&NAME = "Jane";
&IND = &AABYNAME.Find(&NAME);
```

See Also

Chapter 7, Array Class , Replace, page 648, Chapter 7, Array Class , Substitute, page 654.

Get

Syntax

Get(*index*)

Description

Use the **Get** method to return the *index* element of an array. This method is used with the Java PeopleCode functions, instead of using subscripts (which aren't available in Java.)

Using this method is the same as using a subscript to return an item of an array. In the following example, the two lines of code are identical:

```
&Value = &MyArray[8];

&value = &MyArray.Get(8);
```

Parameters

Parameter	Description
index	The array element to be accessed.

Returns

An element in an array.

See Also

Chapter 7, Array Class , Set, page 650, Chapter 16, Java Class, page 959.

Join

Syntax

Join([*separator* [, *arraystart*, *arrayend*]])

Description

The **Join** function *converts* the array that is executing the method into a string by converting each element into a string and *joining* these strings together, separated by *separator*.

Note. Join does *not* join two arrays together.

Each array or subarray to be joined is preceded by the string given by *arraystart* and followed by the string given by *arrayend*. If the given array is multi-dimensional, then (logically) each subarray is first joined, then the resulting strings are joined together.

Parameters

Parameter	Description
separator	Specifies what the elements in the resulting string should be separated with in the resulting string. *Separator* is set by default to a comma (",").
arraystart	Specifies what each array or subarray to be joined should be preceded with in the resulting string. a*rraystart* is set by default to a left parenthesis ("(").
arrayend	Specifies what each array or subarray to be joined should be followed by in the resulting string. a*rrayend* is set by default to a right parenthesis (")").

Returns

A string containing the converted elements of the array.

Example

The following example:

```
Local array of array of number &AAN;

&AAN = CreateArray(CreateArray(1, 2), CreateArray(3, 4), 5);
&STR = &AAN.Join(", ");
```

produces in &STR the string:

```
((1, 2), (3, 4), 5)
```

See Also

Chapter 1, PeopleCode Built-in Functions, Split , page 441.

Next

Syntax

Next(&*index*)

Description

The **Next** method increments the given index variable. It returns true if and only if the resulting index variable refers to an existing element of the array. **Next** is typically used in the condition of a WHILE clause to process a series of array elements up to the end of the array.

&index must be a variable of type integer, or of type Any initialized to an integer, as Next attempts to update it.

If you want to start from the first element of the array, start Next with an index variable with the value zero. The first thing Next does is to increment the value by one.

Parameters

Parameter	Description
&index	The array element where processing should start. *&index* must be a variable of type integer, or of type Any initialized to an integer, as **Next** attempts to update it.

Returns

True if the resulting index refers to an existing element of the array, False otherwise.

Example

Next can be used in a **While** loop to iterate through an array in the following manner:

```
&INDEX = 0;
While &A.Next(&INDEX)
   /* Process &A[&INDEX] */
End-While;
```

In the following code example, &BOTH is a two-dimensional array. This example writes the data from each subarray in &BOTH into a different file.

```
&I = 0;
While &BOTH.Next(&I)
   &J = 1;
   &STRING1 = &BOTH[&I][&J];
   &MYFILE1.writeline(&STRING1);
   &J = &J + 1;
   &STRING2 = &BOTH[&I][&J];
   &MYFILE2.writeline(&STRING2);
End-While;
```

See Also

Chapter 7, Array Class , Len, page 656.

Pop

Syntax

Pop()

Description

Pop removes the last element from the array and returns its value.

Parameters

None.

Returns

The value of the last element of the array. If the last element is a subarray, the subarray is returned.

Example

Pop can be used with the **Push** method to use an array as a stack. To put values on the end of the array, use **Push**. To take the values back off the end of the array, use **Pop**.

Suppose we have a two-dimensional array &SUBPARTS which gives the subparts of each part of some assemblies. Each row (subarray) of &SUBPARTS starts with the name of the part, and then has the names of the subparts. Assuming there are no "loops" in this data, the following code puts all the subparts, subsubparts, and so on, of the part given by &PNAME into the array &ALLSUBPARTS, in "depth first order" (that is, subpart1, subparts of subpart1, …, subpart2, subparts of subpart2, …). We stack the indexes into &SUBPARTS when we want to go down to the subsubparts of the current subpart.

```
Local array of array of string &SUBPARTS;
Local array of string &ALLSUBPARTS;
Local array of array of number &STACK;
Local array of number &CUR;
Local string &SUBNAME;

/* Set the ALLSUBPARTS array to an empty array of string.  */

&ALLSUBPARTS = CreateArrayRept("dummy", 0);

/* Start with the part name.  */

&STACK = CreateArray(CreateArray(&SUBPARTS.Find(&PNAME), 2));
While &STACK.Len > 0
   &CUR = &STACK.Pop();
   If &CUR[1] <> 0 And
         &CUR[2] <= &SUBPARTS[&CUR[1]].Len Then

      /* There is a subpart here.  Add it.  */
      &SUBNAME = &SUBPARTS[&CUR[1], &CUR[2]];
      &ALLSUBPARTS.Push(&SUBNAME);

      /* Tour its fellow subparts later.  */
      &STACK.Push(CreateArray(&CUR[1], &CUR[2] + 1));

      /* Now tour its subsubparts.  */
      &STACK.Push(CreateArray(&SUBPARTS.Find(&SUBNAME), 2));

   End-If;
End-While;
```

See Also

Chapter 7, Array Class , Push, page 646, Chapter 7, Array Class , Replace, page 648, Chapter 7, Array Class , Shift, page 651, Chapter 7, Array Class , Unshift, page 655.

Push

Syntax

Push(*paramlist*)

Where *paramlist* is an arbitrary-length list of values in the form:
value1 [, *value2*] ...

Description

Push adds the values in *paramlist* onto the *end* of the array executing the method. If a value is not the correct dimension, it is flattened or promoted to the correct dimension first, then the resulting values are added to the end of the array.

Considerations Using Arrays With Object References

This method only adds an element to the end of an array. It does not clone or otherwise deep-copy the parameters. For example, if you are adding a reference to an object, Push just adds a reference to the object at the end of the array. This is similar to an assignment. It is *not* making a copy of the object. The following code snippet only puts a reference to the same record onto the end of the array.

```
While &SQL.Fetch(&Rec);
&MYARRAY.Push(&Rec);
...
End-While;
```

Even though the array is growing, all the elements point to the same record. You have only as many standalone record objects as you create. The following code snippet creates new standalone records, so each element in the array points to a new object:

```
local Record &FetchedRec = CreateRecord(Record.PERSONAL_DATA);

While &SQL.Fetch(&FetchedRec)
     &MYARRAY.Push(&FetchedRec);
     &FetchedRec = CreateRecord(Record.PERSONAL_DATA);
End-While;
```

Parameters

Parameter	Description
paramlist	An arbitrary-length list of values, separated by commas.

Returns

None.

Example

The following example loads an array with data from a database table.

```
Local array of record &MYARRAY;
Local SQL &SQL;

&I = 1;
&SQL = CreateSQL("Select(:1) from %Table(:1) where EMPLID like '8%'", &REC);
While &SQL.Fetch(&REC);
   &MYARRAY.Push(CreateRecord(RECORD.PERSONAL_DATA));
   &I = &I + 1;
```

```
      &REC.CopyFieldsTo(&MYARRAY[&I]);
   End-While;
```

See Also

Chapter 7, Array Class , Pop, page 645, Chapter 7, Array Class , Replace, page 648, Chapter 7, Array Class , Shift, page 651, Chapter 7, Array Class , Unshift, page 655, Chapter 7, Array Class , Using Flattening and Promotion , page 640, *PeopleCode Developer's Guide*, Understanding Objects and Classes in PeopleCode, Assigning Objects.

Replace

Syntax

Replace(*start, length*[, *paramlist*])

Where *paramlist* is an arbitrary-length list of values in the form:
value1 [, *value2*] ...

Description

Replace replaces the *length* elements starting at *start* with the given values, if any. If *length* is zero, the insertion takes place *before* the element indicated by *start*. Otherwise, the replacement starts with *start* and continues up to and including *length*, replacing the existing values with *paramlist*.

If a negative number is used for *start*, it indicates the starting position relative to the last element in the array, such that −1 indicates the position just *after* the end of the array. To insert at the end of the array (equivalent to the **Push** method), use a *start* of −1 and a *length* of 0.

If a negative number is used for *length*, it indicates a length measuring downward to lower indexes. Both flattening and promotion can be applied to change the dimension of the supplied parameters to match the elements of the given array.

Similar to how the built-in function **Replace** is used to update a string, the Replace method is a general way to update an array, and can cause the array to grow or shrink.

See Chapter 7, Array Class , Using Flattening and Promotion , page 640.

Using Replace to Remove an Element

You can use the Replace method to remove an element from an array. Just specify the item you want replaces, with *length* equal to one.

The following example removes the item from &Index:

```
&Array.Replace(&Index, 1);
```

Parameters

Parameter	Description
start	Specifies where to start replacing the given elements in the array. If a negative number is used for *start*, it indicates the starting position relative to the last element in the array.
length	Specifies the number of elements in the array to be replaced.
paramlist	Specifies values to be used to replace existing values in the array. This parameter is optional.

Returns

None.

Example

For example, given the following array:

```
Local array of string &AS;

&AS = CreateArray("AA", "BB", "CC");
```

After executing the next code, the array &AN will contain four elements, ZZ, YY, BB, CC:

```
&AS.Replace(1, 1, "ZZ", "YY");
```

After executing the next code, the array &AN will contain three elements, ZZ, MM, CC:

```
&AS.Replace(2, 2, "MM");
```

After executing the next code, the array &AN will contain three elements, ZZ, OO, CC.

```
&AS.Replace( - 2, - 1, "OO");
```

Local Name	Local Value
⊟ &AS	Array
└ Dimension	1
└ Len	3
└ [1]	ZZ
└ [2]	OO
└ [3]	CC

&AS expanded in PeopleCode debugger

See Also

Chapter 7, Array Class , Substitute, page 654, Chapter 7, Array Class , Find, page 642.

Reverse

Syntax

Reverse()

Description

Reverse reverses the order of the elements in the array.

If the array is composed of subarrays, **Reverse** reverses only the elements in the super-array, it doesn't reverse all the elements in the subarrays. For example, the following:

```
&AN = CreateArray(CreateArray(1, 2), CreateArray(3, 4), ⇒
CreateArray(5, 6)).reverse();
```

results in &AN containing:

```
((5,6), (3,4), (1,2)
```

Parameters

None.

Returns

None.

Example

Suppose you had the following array.

```
Local Array of Sting &AS;

&AS = CreateArray("R", "O", "S", "E");
```

If you executed the Reverse method on this array, the elements would be ESOR.

See Also

Chapter 7, Array Class , Sort, page 651.

Set

Syntax

Set(*index*)

Description

Use the **Set** method to set the value of the *index* element of an array. This method is used with the Java PeopleCode functions, instead of using subscripts (which aren't available in Java.)

Using this method is the same as using a subscript to reference an item of an array. In the following example, the two lines of code are identical:

```
&MyArray[8] = &MyValue;

&MyArray.Set(8) = &MyValue;
```

Parameters

Parameter	Description
index	The array element to be accessed.

Returns

None.

See Also

Chapter 7, Array Class , Get, page 643, Chapter 16, Java Class, page 959.

Shift

Syntax

Shift()

Description

Shift removes the first element from the array and returns it. Any following elements are "shifted" to an index of one less than they had.

Parameters

None.

Returns

Returns the value of the first element of the array. If the first element is a subarray, the subarray is returned.

Example

```
For &I = 1 to &ARRAY.Len;
   &ITEM = &ARRAY.Shift;
   /* do processing */
End-For;
```

See Also

Chapter 7, Array Class , Pop, page 645, Chapter 7, Array Class , Push, page 646, Chapter 7, Array Class , Replace, page 648, Chapter 7, Array Class , Unshift, page 655.

Sort

Syntax

Sort(*order*)

Description

Sort rearranges the elements of the array executing the method into an order.

The type of sort done by this function, that is, whether it is a linguistic or binary sort, is determined by the **Sort Order Option** on the PeopleTools Options page.

If the array is one-dimensional, the elements are arranged in either ascending or descending order.

The type of the first element is used to determine the kind of comparison to be made. Any attempt to sort an array whose elements are not all of the same type results in an error.

If *order* is "A", the order is ascending; if it is "D", the order is descending. The comparison between elements is the same one as if done using the PeopleCode comparison operators (<, >, =, and so on.)

Note. If you execute this method on a server, the string sorting order is determined by the character set and localization of the server.

If the array is two-dimensional, the subarrays are arranged in order by the first element of each subarray. Sorting an array whose subarrays have different types of first elements will result in an error. The comparison is done by using the PeopleCode comparison operators (<, >, =, and so on.)

Note. This method works with arrays that have only one or two dimensions. You receive a runtime error if you try to use this method with an array that has more than two dimensions.

Parameters

Parameter	Description
order	Specifies whether the array should be sorted in ascending or descending order. Values for *order* are: • **A**: Ascending • **D:** Descending

Returns

None.

Example

The following example changes the order of the elements in array &A to be ("Frank", "Harry", "John").

```
&A = CreateArray("John", "Frank", "Harry");

&A.Sort();
```

The following example changes the order of the elements in array &A to be (("Frank", 1957), ("Harry", 1928), ("John", 1952)).

```
&A = CreateArray(CreateArray("John", 1952), CreateArray("Frank", 1957), ⇒
CreateArray("Harry", 1928));
```

Local Name	Local Value
&A	Array
Dimension	2
Len	3
[1]	
Dimension	1
Len	2
[1]	John
[2]	1952
[2]	
Dimension	1
Len	2
[1]	Frank
[2]	1957
[3]	
Dimension	1
Len	2
[1]	Harry
[2]	1928

&A expanded in PeopleCode debugger

```
&A.Sort("A");
```

Local Name	Local Value
&A	Array
Dimension	2
Len	3
[1]	
Dimension	1
Len	2
[1]	Frank
[2]	1957
[2]	
Dimension	1
Len	2
[1]	Harry
[2]	1928
[3]	
Dimension	1
Len	2
[1]	John
[2]	1952

&A expanded in PeopleCode debugger, showing code results

See Also

Chapter 7, Array Class , Reverse, page 649, *PeopleSoft Server Tools Administration*, PeopleTools Utilities, PeopleTools Options.

Subarray

Syntax

Subarray(*start, length*)

Description

Subarray creates a new array from an existing one, taking the elements from *start* for a total of *length*. If *length* if omitted, all elements from *start* to the end of the array are used.

If the array is multi-dimensional, the subarrays of the created array are *references* to the same subarrays from the existing array. This means if you make changes to the original subarrays, the referenced subarrays are also changed. To make distinct subarrays, use the Clone method.

Parameters

Parameter	Description
start	Specifies where in the array to begin the subarray.
length	Specifies the number of elements in the array to be part of the subarray.

Returns

An array object.

Example

To make a distinct array from a multi-dimensional array, use the following:

```
&A = &AAN.Subarray(1, 2).Clone();
```

See Also

Chapter 7, Array Class , Clone, page 641.

Substitute

Syntax

Substitute(*old_val, new_val*)

Description

Substitute replaces every occurrence of a value found in an array with a new value. To replace an element that occurs in a specific location in an array, use Replace.

If the array is one-dimensional, **Substitute** replaces every occurrence of the *old_val* in the array with *new_val*.

If the array is two-dimensional, **Substitute** replaces every subarray whose first element is equal to *old_val*, with the subarray given by *new_val*.

Note. This method works with arrays that have only one or two dimensions. You receive a runtime error if you try to use this method with an array that has more than two dimensions.

Parameters

Parameter	Description
old_val	Specifies the existing value in the array to be replaced.
new_val	Specifies the value with which to replace occurrences of *old_val*.

Returns

None

Example

The following example changes the array &A to be ("John", "Jane", "Hamilton").

```
&A = CreateArray();
&A[1] = "John";
&A[2] = "Jane";
&A[3] = "Henry";
&A.Substitute("Henry", "Hamilton");
```

The following example changes the array &A to be (("John", 1952), ("Jane", 1957), ("Hamilton", 1971), ("Frank", 1961)).

```
&A = CreateArray(CreateArray("John", 1952), CreateArray("Jane", 1957), ⇒
CreateArray("Henry", 1928), CreateArray("Frank", 1961));

&A.Substitute("Henry", CreateArray("Hamilton", 1971));
```

See Also

Chapter 7, Array Class , Find, page 642, Chapter 7, Array Class , Replace, page 648.

Unshift

Syntax

Unshift(*paramlist*)

Where *paramlist* is an arbitrary-length list of values in the form:
value1 [*, value2*] ...

Description

Unshift adds the given elements to the *start* of the array. Any following elements are moved up to indexes that are larger by the number of values moved. Flattening and Promotion are used to change the dimension of the supplied parameters to be one less than that of the given array.

Parameters

Parameter	Description
paramlist	Specifies values to be added to the start of the array.

Returns

None.

Example

The following code changes &A to be ("x", "Y", "a", "B", "c").

```
&A = CreateArray("a", "B", "c");

&A.Unshift("x", "Y");
```

See Also

Chapter 7, Array Class , Pop, page 645, Chapter 7, Array Class , Push, page 646, Chapter 7, Array Class , Replace, page 648, Chapter 7, Array Class , Shift, page 651, Chapter 7, Array Class , Using Flattening and Promotion , page 640.

Array Class Properties

In this section, we discuss the array class properties.

Dimension

Description

Dimension is the number of "Array" type names from the declaration of the array, also called subarrays. This property returns a number.

This property is read-only.

Example

The following example sets &DIM to 2.

```
Local Array of Array of Number &AAN;

&DIM = &AAN.Dimension;
```

Len

Description

Len is the current number of elements in the array. This property can be updated. Setting it to a negative value results in an error.

If this property is set to a smaller (nonnegative) number than its current value, the array is truncated to that length, discarding any elements whose indexes are larger than the given new length.

If this property is set to a number larger than its current value, the array is extended to the new length. Any new elements are set to a default value based on the element type of the array.

This property is read-write.

Example

The following is a test of whether an array is empty:

```
If &ARR.Len = 0 then

/* &ARR is empty.  */

End-If;
```

CHAPTER 8

Business Interlink Class

This chapter provides an overview of Business Interlink class and discusses the following topics:

- Using the Interlink object
- Deciding which methods to use
- Using the incoming Business Interlink methods and properties
- Understanding the state of an Interlink object
- Using Business Interlink with Application Engine
- Using the Data Type of a Business Interlink object
- Understanding the scope of a Business Interlink object
- Business Interlink class built-in function
- Business Interlink class reference
- Configuration parameters

Understanding Business Interlink Class

The PeopleSoft Business Interlink framework provides a gateway for PeopleSoft applications to the services of any external system. This framework enables any PeopleSoft component (that is, a page, an Application Engine program, and so on) to integrate with any external system in near real-time and batch modes.

This framework enables a PeopleCode program to map the input and outputs of a Business Interlink definition to PeopleCode variables and record fields, then, using an Interlink object, call the Interlink plug-in, which in turn calls the external system, passes the information to the external system, and returns values to the PeopleCode program.

This documentation describes only the Interlink object portion of the PeopleSoft Business Interlink framework.

Each Interlink object is based on a Business Interlink definition, which is created in PeopleSoft Application Designer. An Interlink object can be based on only a single Business Interlink definition.

See Also

PeopleTools PeopleBook: Peoplesoft Business Interlink Application Developer Guide

Using the Interlink Object

After you instantiate an Interlink object, use the Interlink object to:

- Populate the input buffer
- Send the data to the Interlink plug-in (by using the **Execute** method)
- Fetch the outputs from the buffer
- Check for status and error messages

After the Business Interlink object is instantiated, you can assign values from constants, PeopleSoft variables, or record fields to the inputs of that Business Interlink object.

When you execute the Business Interlink object, it loads the appropriate Business Interlink Plug-in and passes itself to that Business Interlink Plug-in. The Business Interlink Plug-in processes the input data, passing the input values of the Business Interlink object to the external system and then fills the output values of the Business Interlink object (if there are outputs).

Deciding Which Methods to Use

After you create your Business Interlink definition, you must use PeopleCode to instantiate an Interlink object and execute the Business Interlink plug-in. This PeopleCode can be long and complex. Rather than write it directly, you can drag and drop the Business Interlink definition from the PeopleSoft Application Designer Project View into an open PeopleCode edit pane. PeopleSoft Application Designer analyzes the definition and generates initial PeopleCode as a template, which you can modify.

In this section, we discuss how to:

- Execute the Business Interlink object.
- Support batch input and output.
- Support rowsets.
- Use the flat table methods.
- Support dynamic output.
- Use hierarchical data (BIDocs).

See Also

PeopleTools PeopleBook: Peoplesoft Business Interlink Application Developer Guide

Executing the Business Interlink Object

In most cases, you must use the **Execute** method to execute the Business Interlink object. However, for bulk input, you can use the **BulkExecute** method instead.

The **Execute** and **BulkExecute** methods return a value you can use for status and error checking.

Supporting Batch Input and Output

The methods discussed in this section, except for BulkExecute, add input values to the Business Interlink object one set, or row, at a time. The call to the Business Interlink Plug-in occurs only once. All the input data is passed with the single **Execute**. All output is returned as batch as well. The methods then get the output values one set, or row, at a time.

If you're sending a large amount of data to the input buffers, instead of adding one input row at a time, you might write the data to a staging table, then use the **BulkExecute** method. This method automatically executes; that is, you don't have to use the Execute method. It also automatically fills the output record specified with the method with all the output values in every row in the output buffer if you've specified an output record.

Supporting Rowsets

If your data is mapped into rowsets, you may want to use the **InputRowset** method. This method takes a standard rowset object to populate the inputs for the Business Interlink object. You can use the **FetchIntoRowset** method to repopulate the rowset with new data.

Using the Flat Table Methods

If your data is in a flat table structure, you can use the flat table methods. **AddInputRow** adds rows of input to the Business Interlink object; **FetchNextRow** fetches rows of output from the Business Interlink object.

Supporting Dynamic Output

A Business Interlink can have dynamic output, meaning that the outputs for a Business Interlink object are changeable in data type or number of outputs.

Business Interlinks supplies a set of methods to support dynamic output. These methods enable you to interrogate the output buffer programmatically to determine the number of fields (columns), their types, and their values.

The following methods support dynamic output:

- GetFieldCount
- GetFieldType
- GetFieldValue
- MoveFirst
- MoveNext

Use the **MoveFirst** method to move to the first column, first row of the output buffer, and within a loop, use the **MoveNext** method to move to each row on the output buffer.

Within a **MoveFirst** (or **MoveNext**) loop, use the **GetFieldCount** method to get the number of columns in the output buffer, which is also the number of outputs for this Business Interlink object. Then you can extract the outputs from the buffer in a loop.

Within the **GetFieldCount** loop, use **GetFieldName** to get the name of the output, **GetFieldValue** to get the value of the output (which will be returned as a string), and **GetFieldType** to get the type of the output (if the output is not a string type, you will convert it to this type).

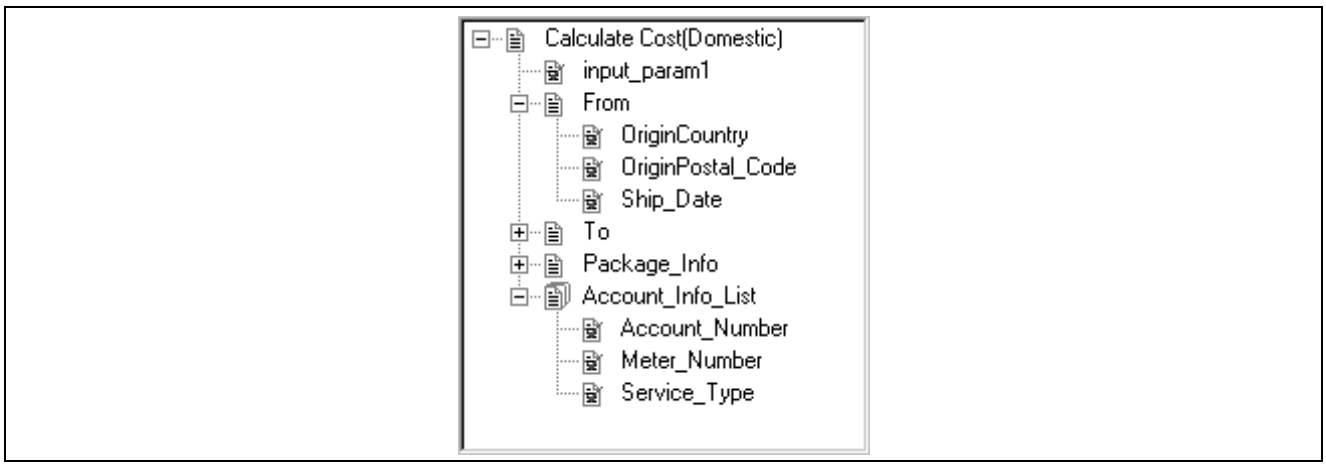
See Chapter 8, Business Interlink Class , GetFieldCount, page 691; Chapter 8, Business Interlink Class , GetFieldType, page 692; Chapter 8, Business Interlink Class , GetFieldValue, page 693; Chapter 8, Business Interlink Class , MoveFirst, page 704 and Chapter 8, Business Interlink Class , MoveNext, page 705.

Using Hierarchical Data (BIDocs)

A Business Interlink can have hierarchical data. Think of the structure as a tree, with a root doc, node docs, and values. The hierarchical data methods and objects are also referred to as **BIDocs**.

Input Structures

The following shows an example of an input structure:



Example input structure

The *root doc* is Calculate Cost(Domestic). A root doc can contain both values and node docs.

The *node docs* are From, To, Package_Info and Account_Info_List. Each node can contain both values and child nodes. Node docs can be further described as either:

- *Simple node docs* have only one set of values for a single instance of a root doc. From, To, Package_Info are all simple node docs.
- *List node docs* can contain more than one set of values for a single instance of a root doc. Account_Info_List is a list node doc.

The **values** in the From node are OriginCountry, OriginPostal_Code, and Ship_Date. The value within the root node is input_param1. Notice that there are values both within the root doc and within node docs.

Business Interlinks support hierarchical input structures with the following methods:

- GetInputDocs
- AddDoc
- AddValue
- AddNextDoc

The **GetInputDocs** method returns a reference to the root doc of an input structure. From the previous example, it returns a reference to Calculate Cost(Domestic).

Use the **AddDoc** method to access the node docs. From the previous example, you would use AddDoc to access the From, To, Package_Info, and Account_Info_List node docs. If any of these nodes contained nodes, you could use AddDoc to access those as well.

Use the **AddValue** method to set values. From the previous example, you would use AddValue to set the value for input_param1, OriginCountry, OriginPosta_Code, and Ship_date. You must call AddDoc on a node before you can call AddValue for its values.

Use the **AddNextDoc** method to access the following:

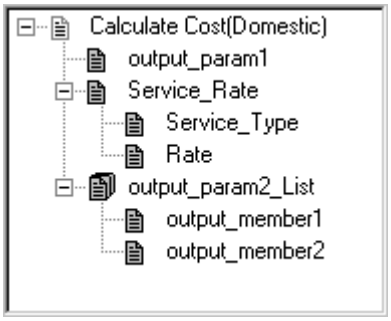
- If a node doc is a list, that is, it can contain more than one set of values, use AddNextDoc to reference the next set of values.
- To add another copy of the entire input structure, use AddNextDoc to return a reference to the next root doc.

The following code example sets values for the node doc From, which is a simple node doc. It also sets the values for Account_Info_List, which is a list node doc.

```
&Calc_Input = &QE_FEDEX_COST.GetInputDocs("");

&FromDoc = &Calc_Input.AddDoc("From");
&ret = &FromDoc.AddValue("OriginCountry", "United States");
&ret = &FromDoc.AddValue("OriginPostal_Code", &ORIGIN);
&ret = &FromDoc.AddValue("Ship_Date", &SHIPDATE);

&Account_Doc = &Calc_Input.AddDoc("Account_Info_List");
&ret = &Account_Doc.AddValue("Account_Number", "CT-8001");
&ret = &Account_Doc.AddValue("Meter_Number", &METER);
&ret = & Account_Doc.AddValue("Service_Type", &MODE);

/* add next set of values in list */

&ret = &Calc_Input.AddNextDoc();
&ret = &Account_Doc.AddValue("Account_Number", "CT-8002");
&ret = &Account_Doc.AddValue("Meter_Number", &METER);
&ret = &Account_Doc.AddValue("Service_Type", &MODE);
```

Output Structures

The following shows an example of an output structure:



Example output structure

The *root doc* is Calculate Cost(Domestic).

The *node docs* are Service_Rate and output_param2_List. Each node can contain both values and child nodes. Node docs can be further be described as either:

- *Simple node docs* have only one set of values for a single instance of a root doc. Service_Rate is a simple node doc.
- *List node docs* can contain more than one set of values for a single instance of a root doc. output_param2_List is a list node doc.

The **values** in the Service_Rate node are Service_Type, Rate, and in the output_param2_List node, output_member1 and output_member2, and within the root node, output_param1.

Business Interlinks support hierarchical output structures with the following methods:

- GetOutputDocs
- GetDoc
- GetNextDoc
- GetPreviousDoc
- GetStatus
- GetValue
- GetCount
- MoveToDoc

The **GetOutputDocs** method returns a reference to the root doc of an output structure. From the previous example, it returns a reference to Calculate Cost(Domestic).

Use the **GetDoc** method to access node docs. From the previous example, you would use GetDoc to access the Service_Rate or output_param2_List node docs. If any of these nodes contain nodes, you use GetDoc to access those as well.

Use the **GetValue** method to retrieve the values. From the previous example, you would use GetValue to retrieve the values for output_param1, and for the Service_Rate node, to get the values Service_Type, Rate, output_member1, and output_member2. You must call GetDoc on a node before you can call GetValue for its values.

Use the **GetNextDoc** (or **GetPreviousDoc**) method to access the following:

- A reference to the next (or previous, respectively) root doc.
- If a node doc is a list, that is, can contain more than one set of values, use GetNextDoc to reference the next node doc in the list (or GetPreviousDoc to access the previous node doc in the list.)

Use the **GetCount** method to return either the number of docs in a list node doc, or the number of root docs. In the example, you can count the number of Calculate Cost(Domestic) nodes or the number of output_param2_list nodes.

Use the **MoveToDoc** method to move to a particular doc at the root level or to a list node doc. In the example, you can move to a Calculate Cost(Domestic) node or to an output_param2_list_node.

The following code example gets values for the node docs Service Rate, which is a simple node doc. It also sets the values for output_param2_List, which is a list node doc.

```
&Calc_Input = &QE_FEDEX_COST.GetOutputDocs("");

&Service_Rate_Doc = &Calc_Input.GetDoc("Service_Rate");
&ret = &Service_Rate_Doc.GetValue("Service_Type", "Overnight");
&ret = &Service_Rate_Doc.GetValue("Rate", "50.00");

&Out_Param_Doc = &Calc_Input.GetDoc("output_param2_List");
&ret = &Out_Param_Doc.GetValue("output_member1", "value1");
&ret = &Out_Param_Doc.GetValue("output_member2", "value2");

/* get next set of values in list */

&Account_Doc = &Out_Param_Doc.AddNextDoc();
&ret = &Out_Param_Doc.GetValue("output_member1", "value3");
&ret = &Out_Param_Doc.GetValue("output_member2", "value4");
```

Using the Incoming Business Interlink Methods and Properties

The incoming Business Interlink methods enable you to parse an XML request and build an XML response.

The incoming Business Interlink uses a set of methods and functions.

The **GetContentBody** Request object method converts the request content into an XML string. You can then use this string with the **GetBiDoc** function to create a BiDocs structure from it that is the same shape and contains the same data as the XML document contained in the XML string.

After you have the BiDocs structure containing the XML request, you can:

- Use the **GetNode** method to get one of the XML elements.
- Use the **NodeType** method to get the type of the node (element, processing instruction, comment).
- Use the **NodeName** property to get the name of the element.
- Use the **NodeValue** property to get the value of the element.

You can also use the standard BiDocs methods (such as GetDoc, GetValue) to retrieve information from this BiDocs object.

When an XML element contains attributes, the **AttributeCount** property gets the number of attributes, the **GetAttributeName** method gets the name of an attribute, and the **GetAttributeValue** method gets the value of an attribute.

To build an XML response, you can use the **GetBiDocs** function to create a blank BiDocs structure. To create the XML structure within that BiDocs, use **CreateElement** to create an XML tag, **AddComment** to add an XML comment, **AddAttribute** to add an attribute to an XML tag, and **AddProcessInstruction** to add a processing instruction (the first tag of the XML response).

Use the **GenXMLString** method to create an XML string from the BiDocs structure. Then you can use the Write Response method to write the response to the HTTP response string.

See Also

PeopleTools PeopleBook: Peoplesoft Business Interlink Application Developer Guide

Understanding the State of an Interlink Object

PeopleSoft Business Interlink API that are run on the application server should be stateless, that is, if you want to save information from one call of the Interlink object to the next, you have to do it yourself by writing the relevant information to the database. If you use the **Execute** method more than once within a single PeopleCode event (that is, if you have the **Execute** method in some sort of loop) the state is preserved. After you leave the event, any state associated with the Interlink object is lost.

You should create only one Business Interlink object, that is, you should only use the **GetInterlink** function once. After that, you can load it with data, pass the data to the Interlink plug-in (using Execute) and fetch output data as many times as you need.

Using Business Interlink with Application Engine

In a regular PeopleCode program, you can only declare a Business Interlink object as local. However, in an Application Engine program, you can declare a Business Interlink object as global. Instantiating a Business Interlink object once as a global saves on the significant overhead of reinstantiating a local object for every iteration of PeopleCode called in a loop.

- Global Business Interlink objects can only be used in Application Engine PeopleCode programs because PeopleCode that runs on an application server must be stateless.
- When a restartable Application Engine program abends, global Business Interlink objects that were instantiated before the last checkpoint are automatically reinstantiated at restart. So the object will be available, even though no call has been made to **GetInterlink** in the restarted process. However, the associated Business Interlink data buffers are *not* recovered, so the Application Engine program must be written such that these buffers are empty whenever a checkpoint is taken.
- Business Interlink objects should *not* be declared as global unless they are used in several PeopleCode actions, or in a PeopleCode action that is called in a loop. Only in these instances is the overhead of doing a checkpoint worthwhile.

Using the Data Type of a Business Interlink Object

You should declare a Business Interlink object using the data type Interlink. For example,

```
Local Interlink &MYBI;
```

Declare hierarchical data as type BIDocs. For example:

```
Local BIDocs &OutlistDoc;
```

Note. BIDocs and Interlink objects used in PeopleCode programs run on the application server can be declared only as type Local. You can declare Interlinks as Global only in an Application Engine program.

Understanding the Scope of a Business Interlink Object

A Business Interlink object can be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

Business Interlink Class Built-in Functions

Chapter 1, PeopleCode Built-in Functions, GetBiDoc , page 236

Chapter 1, PeopleCode Built-in Functions, GetInterlink , page 250

Business Interlink Class Methods

In this section, we discuss the Business Interlink class methods. The methods are discussed in alphabetical order.

AddDoc

Syntax

AddDoc(*docname*)

Description

The **AddDoc** method adds a document to an input structure. The added document is an input parameter for a Business Interlink object that is not of simple type (such as integer or string). You must add the document to the input structure before you can add values to its members with **AddValue**.

Parameters

Parameter	Description
docname	The name of the document that **AddDoc** adds to the structure.

Returns

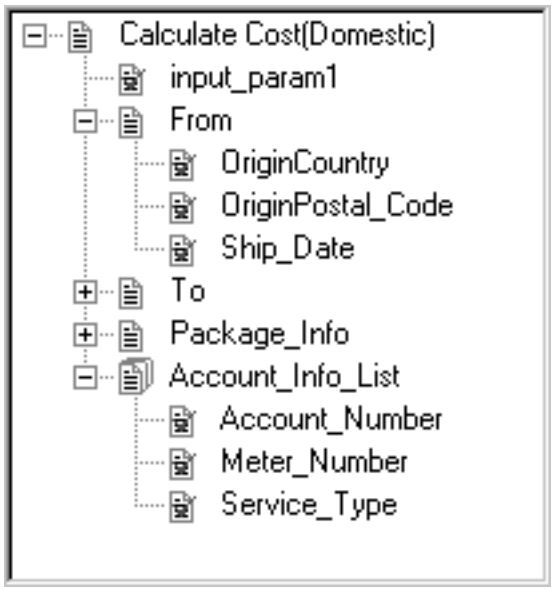
The document added to the BIDocs input document.

Example

In the following example, the input structure for Calculate Cost, or the root level document, is created with the **GetInputDocs** method. (To create, or add, more input structures, use **AddNextDoc**.) The Calculate Cost input parameter From is a document, so the **AddDoc** method adds it to the input document.

```
Local Interlink &QE_FEDEX_COST;
Local BIDocs &CalcCostIn;
Local BIDocs &FromDoc, &ToDoc, &PackageDoc, &AccountDoc;
Local number &ret, &retinput;


&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

&CalcCostIn = &QE_FEDEX_COST.GetInputDocs("");

&ret = &CalcCostIn.AddValue("input_param1","value");
&FromDoc = &CalcCostIn.AddDoc("From");
&ret = &FromDoc.AddValue("OriginCountry", "United States");
&ret = &FromDoc.AddValue("OriginPostal_Code", &ORIGIN);
&ret = &FromDoc.AddValue("Ship_Date", &SHIPDATE);

/* Call AddDoc and AddValue for To, Package_Info, and Account_Info_List   (code⇒
 not shown) */
```

The following shows the input structure for this example. It contains five input parameters: input_param1, From, To, Package_Info, and Account_Info_List. This is a version of the Federal Express plug-in that was modified for this example (input_param1 was added, Account_Info was modified to be a list).



Example input structure

See Also

Chapter 8, Business Interlink Class , GetInputDocs, page 694, Chapter 8, Business Interlink Class , AddValue, page 672, Chapter 8, Business Interlink Class , AddNextDoc, page 670.

AddInputRow

Syntax

AddInputRow(*inputname, value*)

where *inputname* and *value* are in matched pairs, in the form:

inputname1, *value1* [, *inputname2*, *value2*] . . .

Description

The **AddInputRow** method adds a row of input data (*value*) from PeopleCode variables or record fields for the Business Interlink object executing the method. These must be entered in matched pairs, that is, every input name must be followed by its matching value.

Note. The input *name*, not the input path, of the interface definition is used for this method.

There must be an input name for every input parameter defined in the interface definition used to instantiate the Business Interlink object.

If you specify a record field that is not part of the record the PeopleCode program is associated with, you must use *recname.*.*fieldname* for that value.

You can specify default values for every input name in the interface definition (created in PeopleSoft Application Designer.) These values are used if you create a PeopleCode "template" by dragging the interface definition from the Project window in Application Designer to an open PeopleCode editor window.

Parameters

Parameter	Description
inputname	Specify the input name. There must be one *inputname* for every input name defined in the interface definition used to instantiate the Business Interlink object.
value	Specify the value for the input name. This can be a constant, a variable, or a record field. Each *value* must be paired with an *inputname*.

Returns

A Boolean value: True if the input values were successfully added. Otherwise, it returns False.

Example

In the following example, the Business Interlink object name is SRA_ALL_1, and the input name, such as ship_site_name, are being bound to record fields, such as QE_RP_SITENAME or VENDOR_INFO.QE_RP_PROMISEDATE, to variables like &PARTNAME, or to literals, like 10 for quantity.

```
&SRA_ALL_1.AddInputRow("ship_site_name", QE_RP_SITENAME,
   "promise_date", VENDOR_INFO.QE_RP_PROMISEDATE,
   "request_date", QE_RP_ORDERREQDATE,
   "subline_site_name", QE_RP_SITENAME,
   "quantity", 10,
   "part_name", &PARTNAME,
   "site_name", INV_LOCATIONS.QE_RP_SITENAME);
```

See Also

Chapter 8, Business Interlink Class , Execute, page 679, Chapter 1, PeopleCode Built-in Functions, GetInterlink , page 250.
PeopleTools PeopleBook: PeopleTools Application Developer Guide, "Generating the PeopleCode Template"

AddNextDoc

Syntax

```
AddNextDoc()
```

Description

The **AddNextDoc** method adds a document to one of the following levels:

- The root level of the input structure for a Business Interlink object. You create a reference to this level with the **GetInputDocs** method.
- When a document within the input structure is a list, **AddNextDoc** adds another document to the list. If you use **AddNextDoc** on a document that is not a list, **AddNextDoc** fails and returns an error.

Parameters

None.

Returns

This method returns an integer. The values are:

Value	Description
0	eNoError. The method succeeded.
1	eNO_DOCUMENT. The document referenced by this method does not exist.
2	eDOCLIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time results in this error.
3	eDOCUMENT_UNINITIALIZED. Internal error.
4	eNOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.
5	eNOT_LISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.
7	eNOT_BASICTYPE. You tried to perform a GetValue or AddValue upon a parameter that is not of basic type: integer, float, string, time, date, datetime.
8	eNO_DATA. You tried to retrieve data from a document that contained no data.

Example

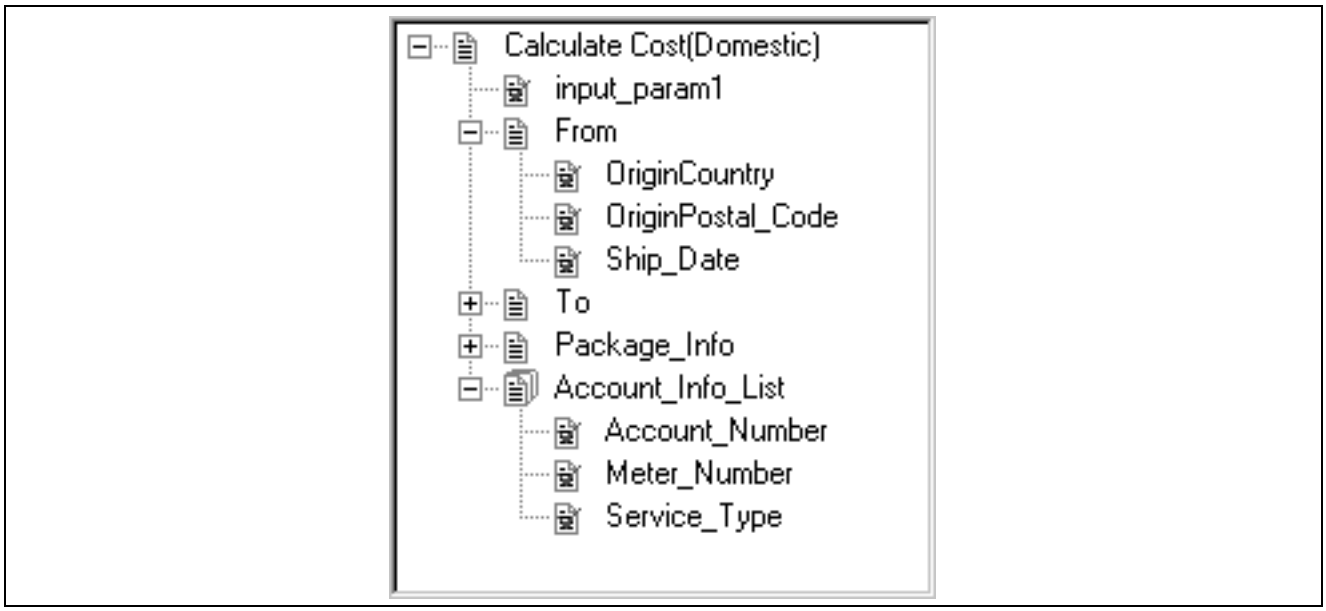
In this example, the input structure for Calculate Cost, or the root level document, is accessed with the **GetInputDocs** method. The **AddNextDoc** method adds another BIDocs input document. The Calculate Cost input parameter Account_Info_List is a document, so the **AddDoc** method adds it to the BIDocs input document. Account_Info_List is also a document list, so **AddNextDoc** method adds another Account_Info_List document.

```
Local Interlink &QE_FEDEX_COST;
Local BIDocs &CalcCostIn;
Local BIDocs &FromDoc, &ToDoc, &PackageDoc, &AccountDoc;
Local number &ret, &retinput;


&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

&CalcCostIn = &QE_FEDEX_COST.GetInputDocs("");

For &n = 1 to &number_of_input_sets
/* Get values for inputs, such as &ORIGIN (code not shown) */

   &ret = &CalcCostIn.AddValue("input_param1","value");
   &FromDoc = &CalcCostIn.AddDoc("From");
   &ret = &FromDoc.AddValue("OriginCountry", "United States");
   &ret = &FromDoc.AddValue("OriginPostal_Code", &ORIGIN);
   &ret = &FromDoc.AddValue("Ship_Date", &SHIPDATE);

   /* Call AddDoc and AddValue for To and Package_Info
      (code not shown) */

   /* Call AddDoc and AddNextDoc for the AccountDoc document list */
   &AccountDoc = &CalcCostIn.AddDoc("Account_Info_List");
   For &m = 1 to &number_of_AccountDocs
      &ret = &AccountDoc.AddValue("Account_Number", &ACCOUNT);
      &ret = &AccountDoc.AddValue("Meter_Number", &METER);
      &ret = &AccountDoc.AddValue("Service_Type", &SVCTYPE);
      &retinput = &AccountDoc.AddNextDoc();
   End-For;

   &retinput = &CalcCostIn.AddNextDoc();

End-For;
```

The following shows the input structure for this example. It contains five input parameters: input_param1, From, To, Package_Info, and Account_Info_List.

From, To, and Package_Info are not lists, so if you try to use **AddNextDoc** with these parameters, such as the following line of code, **AddNextDoc** will fail and return an error message.

```
&retinput = &To.AddNextDoc();
```



Example input structure

See Also

Chapter 8, Business Interlink Class , GetInputDocs, page 694, Chapter 8, Business Interlink Class , AddDoc, page 667, Chapter 8, Business Interlink Class , AddValue, page 672.

AddValue

Syntax

AddValue(*paramname, value*)

Description

The **AddValue** method adds a value to a member of a document within the input structure for a Business Interlink object.

Parameters

Parameter	Description
paramname	The name of the member of the document that is having a value added to it.
value	The value that is added. This can be a constant, a variable, or a record field. The data type can be string, integer, double, float, time, date, or datetime.

Returns

Number: Enum value and Meaning

0: eNoError. The method succeeded.

1: eNO_DOCUMENT. The document referenced by this method does not exist.

2: eDOCLIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time results in this error.

3: eDOCUMENT_UNINITIALIZED. Internal error.

4: eNOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.

5: eNOT_LISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.

7: eNOT_BASICTYPE. You tried to perform a GetValue or AddValue upon a parameter that is not of basic type: integer, float, string, time, date, datetime.

8: eNO_DATA. You tried to retrieve data from a document that contained no data.

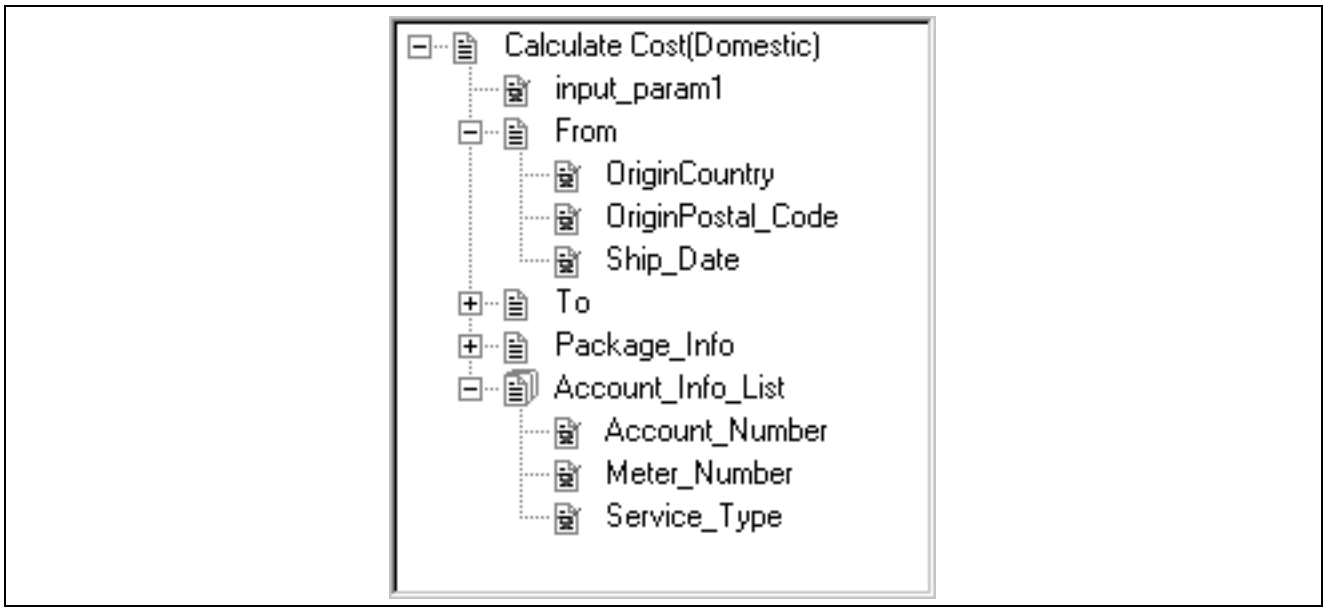
Example

In the following example, the Business Interlink object name is QE_FEDEX_COST.

In the following example, the input structure for Calculate Cost, or the root level document, is accessed with the **GetInputDocs** method. (To create, or add, more input structures, use **AddNextDoc**.) The Calculate Cost input parameter From is a document, so the **AddDoc** method adds it to the input structure. Then the **AddValue** method adds values to each of the From document members.

```
Local Interlink &QE_FEDEX_COST;
Local BIDocs &CalcCostIn;
Local BIDocs &FromDoc, &ToDoc, &PackageDoc, &AccountDoc;
Local number &ret;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

/* Get some values for input, such as &ORIGIN (code not shown) */

&CalcCostIn = &QE_FEDEX_COST.GetInputDocs("");
&ret = &CalcCostIn.AddValue("input_param1","value");

&FromDoc = &CalcCostIn.AddDoc("From");
&ret = &FromDoc.AddValue("OriginCountry", "United States");
&ret = &FromDoc.AddValue("OriginPostal_Code", &ORIGIN);
&ret = &FromDoc.AddValue("Ship_Date", &SHIPDATE);

/* Call AddDoc, AddValue for Package, Account_Info_List, and also call AddNextDoc⇒
 for Account_Info_List (code not shown) */
```

The following shows the input structure for this example. It contains five input parameters: input_param1, From, To, Package_Info, and Account_Info_List.



Example input structure

See Also

Chapter 8, Business Interlink Class , GetInputDocs, page 694, Chapter 8, Business Interlink Class , AddDoc, page 667, Chapter 8, Business Interlink Class , AddNextDoc, page 670.

BulkExecute

Syntax

```
BulkExecute(RECORD.inputrecname [, RECORD.outputrecname] [, {user_
process_inst | user_operid}])
```

Description

The **BulkExecute** method uses the data in the specified record to populate the input buffer, copying **like-named** fields. Then the method executes, and, optionally, fills the record specified by *outputrecname* with data from the Business Interlink output buffer. **BulkExecute** results in significantly faster performance for transactions that process large amounts of data. Instead of adding one input row at a time, then fetching the values one at a time, you might write the data to a staging table, use the **BulkExecute** method and then read the data from the output table. This would be especially effective in Application Engine programs that process sets of data rather than individual rows.

This method assumes that the names of the fields in the record match the names of the inputs (or outputs) defined in the Business Interlink Definition. If there is no field in the *inputrecname* for a Business Interlink input parameter, the parameter's default value is used. If no default is specified, an empty string is passed. You must ensure that this default value is legitimate.

Fields in the output buffer are matched against the fields specified in the output record. You do not have to specify an output record. If you don't specify an output record, the output buffers are *not* be populated.

If you specify an output record, and if you have fields in the output record that are *not* specified as output parameters, they aren't populated with the BulkExecute method. In addition, they shouldn't be set as **NOT NULL**, otherwise inserts fail.

Note. *Before* you use this method, you should flush the record used for output and remove any residual data that might exist in it.

If you specify an output record, and you want to use the output record for error checking, you must add the following fields to your output record:

- RETURN_STATUS
- RETURN_STATUS_MSG

Note. The field names in the output record must match these names exactly.

Then you must mark one or more input parameters as "key fields" in the Business Interlink definition (using the BulkExecute ID check box.) The value of these key fields are copied to the output record, and you can use these fields to match error messages with input (or output) rows.

To order your input (and output) rows, you must add the following column to both the input and output table:

- BI_SEQ_NUM
-

Note. The field name in your input and output records must match this name exactly.

The input rows are read in the order of numbers in BI_SEQ_NUM, and the output rows are generated using the same order number.

This method automatically executes, so you don’t need to use the **Execute** method with this method.

Whether this method halts on execution depends on the setting of the StopAtError configuration parameter. The default value is True, that is, stop if the error number returned is something other than a 1 or 2. This configuration parameter must be set *before* using the **BulkExecute** method.

Parameters

Parameter	Description
RECORD.inputrecname	Specify a record (SQL table) that contains the data to populate the input buffer of the Business Interlink.
RECORD.outputrecname	Specify a record (SQL table) that will hold the data that populates the output buffer of the Business Interlink. This is optional; it is often used to error check the input.
user_process_inst | user_operid	This is an optional parameter that allows *either* different Application Engine programs *or* different clients to populate the same **RECORD.** *outputrecname* at the same time.
	For *user_process_inst*, the parameter takes an integer. **RECORD.** *outputrecname* must have a PROCESS_INSTANCE field. The PROCESS_INSTANCE field is used to identify the Application Engine program that is using this Business Interlink. You can use the %PROCESS_INSTANCE variable to populate *user_process_inst*.
	For *user_operid*, the parameter takes a string. **RECORD.** *outputrecname* must have an OPERID field. The OPERID field is used to identify the client who is using this Business Interlink. You can use the %OPERID variable to populate *user_operid*.

Returns

Number: Meaning

1: The Business Interlink object executes successfully

2: The Business Interlink object failed to execute

3: Transaction failed

4: Query failed

5: Missing criteria

6: Input mismatch

7: Output mismatch

8: No response from server

9: Missing parameter

10: Invalid username

11: Invalid password

12: Invalid server name

13: Connection error

14: Connection refused

15: Timeout reached

16: Unequal lists

17: No data for output

18: Output parameters empty

19: Driver not found

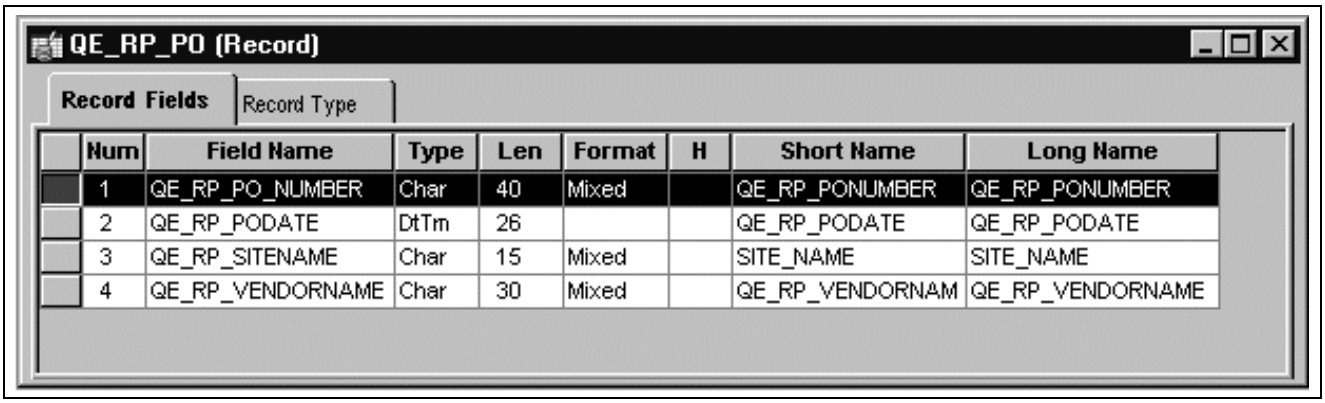
20: Internet connect error

21: XML parser error

22: XML deserialize

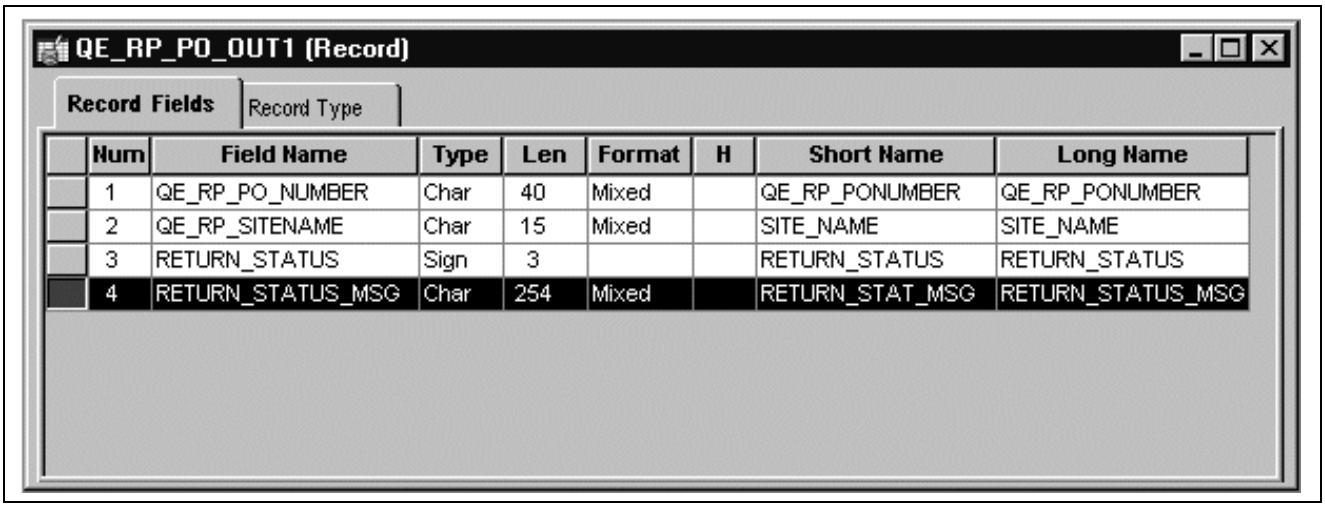
Example

Here are two PeopleSoft records that could be used as input and output records for BulkExecute. The key fields in the input record, which need to have the BulkExecute ID check box checked in PeopleSoft Application Designer, are QE_RP_PO_NUMBER and QE_RP_SITENAME. These key fields are used both for input and output.



Num	Field Name	Type	Len	Format	H	Short Name	Long Name
1	QE_RP_PO_NUMBER	Char	40	Mixed		QE_RP_PONUMBER	QE_RP_PONUMBER
2	QE_RP_PODATE	DtTm	26			QE_RP_PODATE	QE_RP_PODATE
3	QE_RP_SITENAME	Char	15	Mixed		SITE_NAME	SITE_NAME
4	QE_RP_VENDORNAME	Char	30	Mixed		QE_RP_VENDORNAM	QE_RP_VENDORNAME

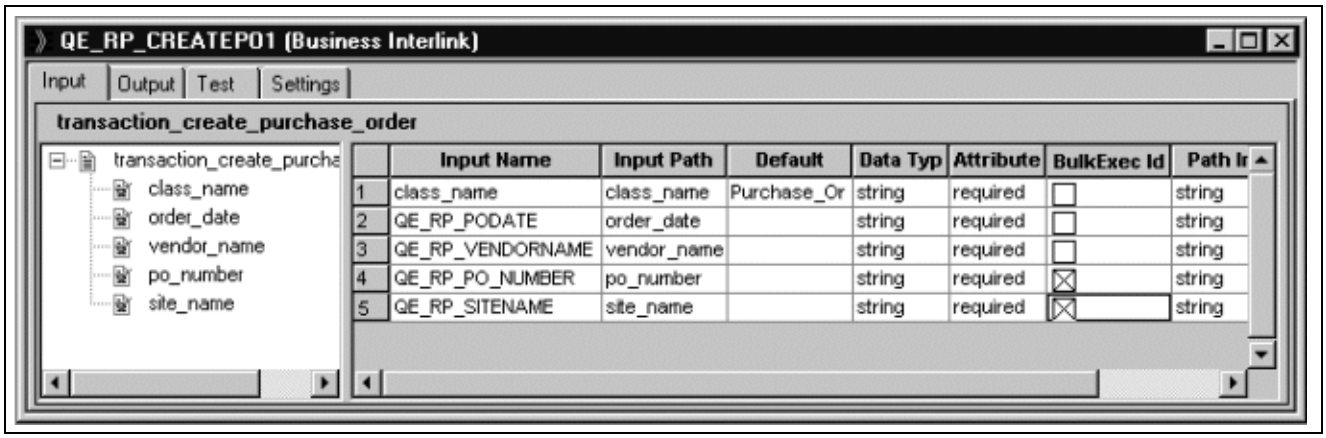
Example input record for BulkExecute



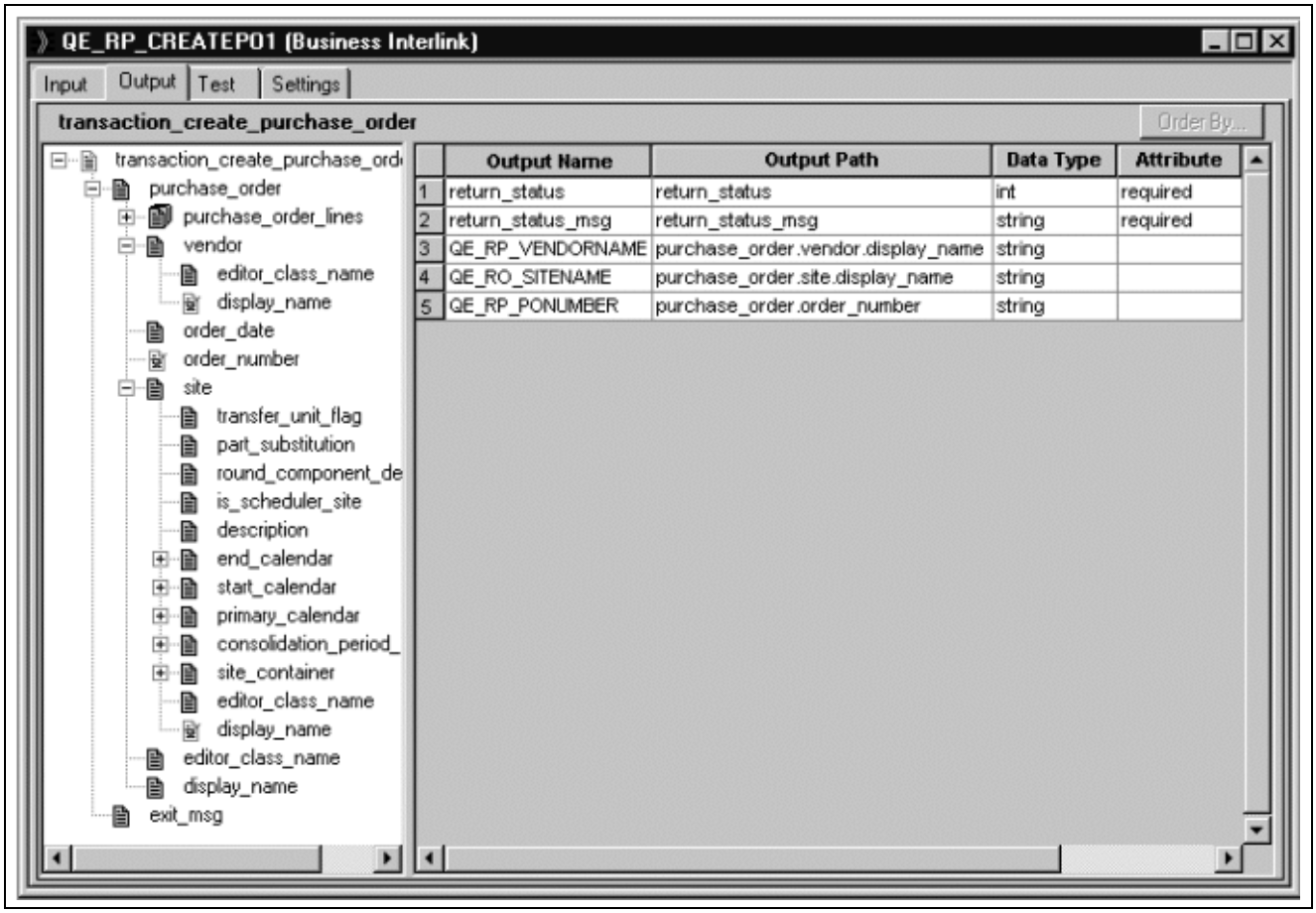
Num	Field Name	Type	Len	Format	H	Short Name	Long Name
1	QE_RP_PO_NUMBER	Char	40	Mixed		QE_RP_PONUMBER	QE_RP_PONUMBER
2	QE_RP_SITENAME	Char	15	Mixed		SITE_NAME	SITE_NAME
3	RETURN_STATUS	Sign	3			RETURN_STATUS	RETURN_STATUS
4	RETURN_STATUS_MSG	Char	254	Mixed		RETURN_STAT_MSG	RETURN_STATUS_MSG

Example output record for BulkExecute

Here are the corresponding inputs and outputs for a Business Interlink that has had its inputs and outputs renamed to match the field names in the records. The inputs and outputs have been renamed where necessary to match the record field names.



Example Business Interlink inputs for BulkExecute



Example Business Interlink inputs for BulkExecute

The PeopleCode program using these records could contain the following code to run **BulkExecute**.

```
Local Interlink &CREATE_PO;
Local number &EXECRSLT;

&CREATE_PO = GetInterlink(INTERLINK.QE_RP_CREATEPO1);

/* The next three lines set configuration parameters, which are not shown in the⇒
 previous examples. */
```

```
&CREATE_PO.SERVER_NAME = "bc1";
&CREATE_PO.RSERVER_HOST = "4.3.4.3";
&CREATE_PO.RSERVER_PORT = "2200";

&EXECRSLT = &CREATE_PO.BulkExecute(RECORD.QE_RP_PO, RECORD.QE_RP_PO_OUT1);
```

See Also

Chapter 8, Business Interlink Class , StopAtError, page 721.

Clear

Syntax

Clear()

Description

The **Clear** method clears the input and output buffers. If you're using Business Interlinks in a batch program, every time after you use the **Execute** method, use the **Clear** method to flush out the input and output buffers.

Parameters

None.

Returns

None.

Example

```
For &n = 1 to x
   &myinterlink.AddInputRow("input1", value1, "input2", value2);
   &myinterlink.Execute();
   &myinterlink.FetchNextRow("output1", value1, "output2", value2);
      /* do processing on data */
   &myinterlink.Clear();
   &n = &n + 1;
End-for;
```

See Also

Chapter 8, Business Interlink Class , Execute, page 679, Chapter 8, Business Interlink Class , BulkExecute, page 674.

Execute

Syntax

Execute()

Description

When an Interlink object executes the **Execute** method, the transaction associated with the Interlink object is executed.

If there is only one row, after appropriate substitution is made, the transaction is executed only once. Otherwise, the data is "batched up" and sent once. You have to call **Execute** only once to execute all the rows of the input buffer.

Generally, you would only use the **Execute** method after using the **AddInputRow** method. If you create a PeopleCode "template" by dragging the Business Interlink definition from the Project window in PeopleSoft Application Designer to an open PeopleCode editor window, the usual order of the execution of methods is shown in the template.

If you’re using Business Interlinks in a batch program, every time after you use the **Execute** method, use the **Clear** method to flush out the input and output buffers.

Whether this method halts on execution depends on the setting of the StopAtError configuration parameter. The default value is True, that is, stop if the error number returned is something other than a 1 or 2. This configuration parameter must be set *before* using the **Execute** method.

Parameters

None.

Returns

Number: Meaning

1: The Interlink object executes successfully

2: The Interlink object failed to execute

3: Transaction failed

4: Query failed

5: Missing criteria

6: Input mismatch

7: Output mismatch

8: No response from server

9: Missing parameter

10: Invalid username

11: Invalid password

12: Invalid server name

13: Connection error

14: Connection refused

15: Timeout reached

16: Unequal lists

17: No data for output

18: Output parameters empty

19: Driver not found

20: Internet connect error

21: XML parser error

22: XML deserialize

Example

```
Local number &EXECRSLT;
.  .  .
&EXECRSLT = &SRA_ALL_1.Execute();
If (&EXECRSLT <> 1) Then
/* The instance failed to execute */
/* Do Error Processing */
End-If;
```

See Also

Chapter 8, Business Interlink Class , BulkExecute, page 674, Chapter 8, Business Interlink Class , Clear, page 679, Chapter 1, PeopleCode Built-in Functions, GetInterlink , page 250.

FetchIntoRecord

Syntax

FetchIntoRecord(**RECORD.***recname*, [, {*user_process_inst* | *user_operid*}])

Description

The **FetchIntoRecord** method copies the data from the output buffer into the specified record (SQL table) copying *like-named* fields. This method assumes that the names of the fields in the record match the names of the outputs defined in the Business Interlink definition.

You can use the FetchIntoRecord method to perform a transaction on a large amount of data that you want to receive from your system. Instead of executing your Business Interlink and then fetching one output row at a time, you might execute your Business Interlink, then use the **FetchIntoRecord** method to write the data returned from the Business Interlink to a staging table.

Note. *Before* you use this method, you should flush the record used for output and remove any residual data that might exist in it.

If your data is hierarchical, that is, in a rowset, and you want to preserve the hierarchy, use **FetchIntoRowset** instead of this method.

To order your output rows, you must add the following column to the record:

- BI_SEQ_NUM
-

This column is populated with a sequence number that corresponds to the order in which each row of the record was written to by the method.

Parameters

Parameter	Description
RECORD.recname	Specify a record (SQL table) that you want to populate with data from the output buffers.
user_process_inst | user_operid	This is an optional parameter that enables *either* different Application Engine programs *or* different clients to populate the same **RECORD.** *outputrecname* at the same time.
	For *user_process_inst*, the parameter takes an integer. **RECORD.** *outputrecname* must have a PROCESS_INSTANCE field. The PROCESS_INSTANCE field is used to identify the Application Engine program that is using this Business Interlink. You can use the %PROCESS_INSTANCE variable to populate *user_process_inst*.
	For *user_operid*, the parameter takes a string. **RECORD.** *outputrecname* must have an OPERID field. The OPERID field is used to identify the client who is using this Business Interlink. You can use the %OPERID variable to populate *user_operid*.

Returns

Number of rows fetched if one or more non-empty rows are returned.

If an empty row is returned, that is, every if field is empty except the return_status and return_status_msg fields, this method returns one of the following error messages:

Number	Meaning
0	No error
1	NoInterfaceObject
2	ParamCountError
3	IncorrectParameterType
4	NoColumnForOutputParm
5	NoColumnForInputParm
6	BulkInsertStartFailed
7	BulkInsertStopFailed
8	CouldNotCreateSelectCursor
9	CouldNotCreateInsertCursor

Number	Meaning
10	CouldNotDestroySelectCursor
11	CouldNotDestroyInsertCursor
12	InputRecordDoesNotExist
13	OutputRecordDoesNotExist
14	ContainEmptyRow
15	SQLExecError
201	FieldTooLarge
202	IgnoreSignNumber
203	ConvertFloatToInt

Example

```
&RSLT = &QE_SM_CONCAT.FetchIntoRecord(RECORD.PERSONAL__VENDR_DATA);

If &RSLT = 0 Then
/* no rows fetched */
Else
   /* do processing */
End-If;
```

See Also

Chapter 8, Business Interlink Class , BulkExecute, page 674.

FetchIntoRowset

Syntax

FetchIntoRowset(*&Rowset*)

Description

The **FetchIntoRowset** method copies the data from the output buffer into the specified rowset, copying *like-named* fields. This method assumes that the names of the fields in the rowset match the names of the outputs defined in the Business Interlink definition, *and* that the structure is the same.

Note. *Before* you use this method, you should flush the rowset used for output and remove any residual data that might exist in it.

Use this method only if you have a hierarchical data structure and you want to preserve the hierarchy. Otherwise, use **BulkExecute** or **FetchIntoRecord**.

Parameters

Parameter	Description
&Rowset	Specify rowset object that you want to populate with data from the output buffers. This must be an existing, instantiated rowset.

Returns

Number of rows fetched if one or more non-empty rows are returned.

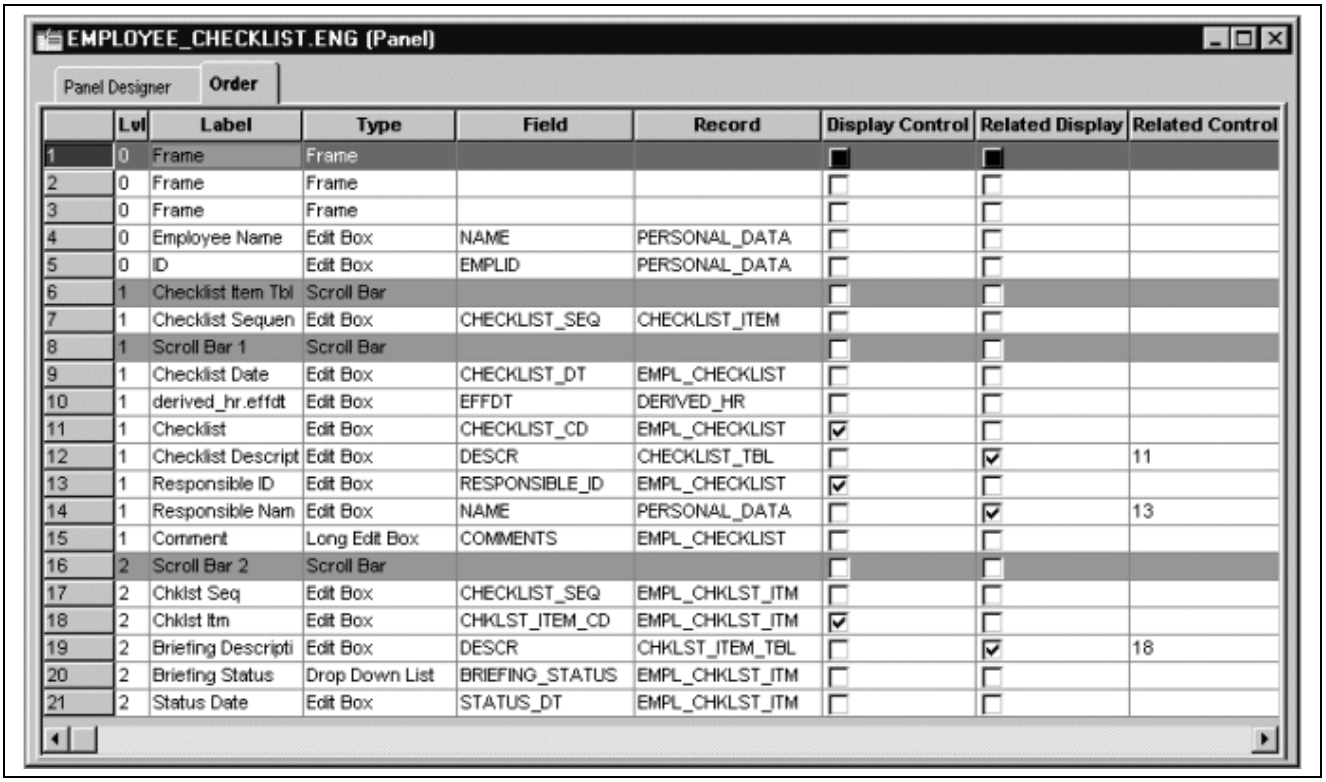
If an empty row is returned, that is, every if field is empty except the return_status and return_status_msg fields, this method returns one of the following error messages:

Value	Description
0	No error
1	NoInterfaceObject
2	ParamCountError
3	IncorrectParameterType
4	NoColumnForOutputParm
5	NoColumnForInputParm
6	BulkInsertStartFailed
7	BulkInsertStopFailed
8	CouldNotCreateSelectCursor
9	CouldNotCreateInsertCursor
10	CouldNotDestroySelectCursor
11	CouldNotDestroyInsertCursor
12	InputRecordDoesNotExist
13	OutputRecordDoesNotExist
14	ContainEmptyRow
15	SQLExecError
201	FieldTooLarge
202	IgnoreSignNumber

Value	Description
203	ConvertFloatToInt

Example

The example uses the rowset on level 1 from the EMPLOYEE_CHECKLIST page. A PeopleCode program running in a field on Level 0 in that page can access the level 1 (child rowset).



	Lvl	Label	Type	Field	Record	Display Control	Related Display	Related Control
1	0	Frame	Frame			■	■	
2	0	Frame	Frame			☐	☐	
3	0	Frame	Frame			☐	☐	
4	0	Employee Name	Edit Box	NAME	PERSONAL_DATA	☐	☐	
5	0	ID	Edit Box	EMPLID	PERSONAL_DATA	☐	☐	
6	1	Checklist Item Tbl	Scroll Bar			☐	☐	
7	1	Checklist Sequen	Edit Box	CHECKLIST_SEQ	CHECKLIST_ITEM	☐	☐	
8	1	Scroll Bar 1	Scroll Bar			☐	☐	
9	1	Checklist Date	Edit Box	CHECKLIST_DT	EMPL_CHECKLIST	☐	☐	
10	1	derived_hr.effdt	Edit Box	EFFDT	DERIVED_HR	☐	☐	
11	1	Checklist	Edit Box	CHECKLIST_CD	EMPL_CHECKLIST	☑	☐	
12	1	Checklist Descript	Edit Box	DESCR	CHECKLIST_TBL	☐	☑	11
13	1	Responsible ID	Edit Box	RESPONSIBLE_ID	EMPL_CHECKLIST	☑	☐	
14	1	Responsible Nam	Edit Box	NAME	PERSONAL_DATA	☐	☑	13
15	1	Comment	Long Edit Box	COMMENTS	EMPL_CHECKLIST	☐	☐	
16	2	Scroll Bar 2	Scroll Bar			☐	☐	
17	2	Chklst Seq	Edit Box	CHECKLIST_SEQ	EMPL_CHKLST_ITM	☐	☐	
18	2	Chklst Itm	Edit Box	CHKLST_ITEM_CD	EMPL_CHKLST_ITM	☑	☐	
19	2	Briefing Descripti	Edit Box	DESCR	CHKLST_ITEM_TBL	☐	☑	18
20	2	Briefing Status	Drop Down List	BRIEFING_STATUS	EMPL_CHKLST_ITM	☐	☐	
21	2	Status Date	Edit Box	STATUS_DT	EMPL_CHKLST_ITM	☐	☐	

EMPLOYEE_CHECKLIST Page order

EMPL_CHECKLIST is the primary database record for the level 1 scrollbar on the EMPLOYEE_CHECKLIST page. The following PeopleCode accesses the level 1 rowset using EMPL_CHECKLIST. The Business Interlink object name is QE_BI_EMPL_CHECKLIST. This Business Interlink object uses the level 1 rowset as its input and its output.

The **InputRowset** method uses this rowset as input for QE_BI_EMPL_CHECKLIST. Then a blank duplicate of the rowset is created with CreateRowset, and then the output of QE_BI_EMPL_CHECKLIST is fetched into the blank rowset with **FetchIntoRowset**.

```
&MYROWSET = GetRowset(SCROLL.EMPL_CHECKLIST);
&ROWCOUNT = &QE_BI_EMPL_CHECKLIST.InputRowset(&MYROWSET);
&RSLT = &QE_BI_EMPL_CHECKLIST.Execute();
/* do some error processing */
&WorkRowset = CreateRowset(&MYROWSET);
&ROWCOUNT = &QE_BI_EMPL_CHECKLIST.FetchIntoRowset(&WorkRowset);

If &ROWCOUNT = 0 Then
/* do some error processing */
Else
```

```
   /* Process the rowset from QE_BI_EMPL_CHECKLIST. Check it for errors. */
   For &I = 1 to &WorkRowset.RowCount
      For &K = 1 to &WorkRowset(&I).RecordCount
         &REC = &WorkRowset(&I).GetRecord(&K);
         &REC.ExecuteEdits();
         For &M = 1 to &REC.FieldCount
            If &REC.GetField(&M).EditError Then
            /* there are errors */
            /* do other processing */
            End-If;
         End-For;
      End-For;
   End-for;
End-if;
```

See Also

Chapter 8, Business Interlink Class , InputRowset, page 702, *PeopleCode Developer's Guide*, Data Buffer Access.

FetchNextRow

Syntax

FetchNextRow(*outputname, value*)

where *outputname* and *value* are in matched pairs, in the form:
outputname1, value1 [*, outputname2, value2*] . . .

Description

After the Business Interlink object executes the method **Execute**, the **FetchNextRow** method can be used to retrieve a row of output and store the values of the output name (*outputname*) *to* PeopleCode variables or record fields (*value)*. These must be entered in matched pairs, that is, every output name must be followed by its matching value.

Note. The output *name,* not the output path of the interface definition is used for this method.

There must be an *outputname* for every output name defined in the interface definition used to instantiate the Business Interlink object.

If you specify a record field that is not part of the record the PeopleCode program is associated with, you must use *recname.fieldname* for that *value*.

You can specify default values for every output name in the interface definition (created in PeopleSoft Application Designer.) These values are used if you create a PeopleCode "template" by dragging the interface definition from the Project window in PeopleSoft Application Designer to an open PeopleCode Editor window.

Parameters

Parameter	Description
outputname	Specify the output name. There must be one *outputname* for every output name defined in the interface definition used to instantiate the Business Interlink object.
value	Specify the value for the output name. This can be a constant, a variable, or a record field. Each *value* must be paired with an *outputname*.

Returns

A Boolean value: True if the row of output parameters was fetched. Otherwise, it returns False.

Example

In the following example, the Business Interlink object name is SRA_ALL_1, and the output names, such as **costs**, are being bound to record fields, such as STR_COST.

```
&RSLT = &SRA_ALL_1.FetchNextRow("costs", &STR_COST,
          "unit_costs", &STR_UNIT_COST,
          "customer_ship_dates", &STR_SHIP_DATE,
          "quantities", &STR_QUANTITY,
          "so_numbers", &STR_SO_NUM,
          "so_names", &STR_SO_NAME,
          "line_numbers", &STR_LINE_NUM,
          "ship_sets", &STR_SHIP_SET,
          "customer_receipt_dates", &STR_RECPT_DATE);
```

See Also

Chapter 8, Business Interlink Class , Execute, page 679, Chapter 1, PeopleCode Built-in Functions, GetInterlink , page 250.

GetCount

Syntax

GetCount(*docname*)

Description

The **GetCount** method returns the number of documents within a document list contained within an output structure for a Business Interlink object.

Parameters

Parameter	Description
docname	The name of the document list.

Returns

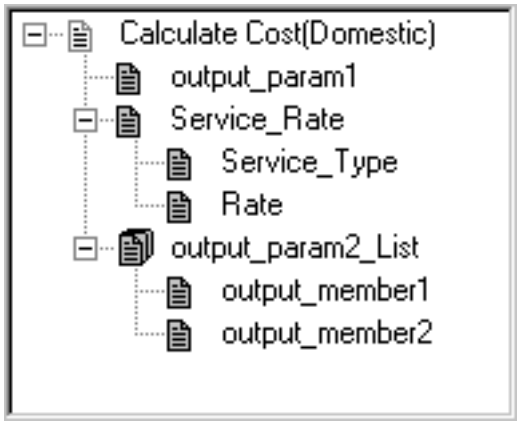
The number of documents in the list.

Example

In the following example, the output structure for Calculate Cost, or the root level document, is accessed with the **GetOutputDocs** method. The Calculate Cost output parameter output_param2_List is a document, so the **GetDoc** method gets it from the output structure. Because output_param2_List is a document list, **GetCount** gets the number of documents in the list.

```
Local Interlink &QE_FEDEX_COST;
Local number &count;
Local BIDocs &CalcCostOut;
Local BIDocs &OutlistDoc;
Local number &ret;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);
// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

/* Call GetValue for output_param1, call GetDoc, GetValue for Service_Rate (code⇒
 not shown) */

&OutlistDoc = &CalcCostOut.GetDoc("output_param2_List");
&count = &CalcCostOut.GetCount("output_param2_List");

&I = 0;
While (&I < &count)
   &ret = &OutlistDoc.GetValue("output_member1", &VALUE1);
   &ret = &OutlistDoc.GetValue("output_member2", &VALUE2);
   If &ret = 0 Then
      /* Process output values */
      &I = &I + 1;
      &retoutput = &OutlistDoc.GetNextDoc();
   End-If;
End-While;
```

The following shows the output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List.



Example output structure

GetDoc

Syntax

GetDoc(*docname*)

Description

The **GetDoc** method gets a document from an output structure. The document is an output parameter for a Business Interlink object that is not of simple type (such as integer or string). You must get the document from the output structure before you can get values from its members with **GetValue**.

You can call **GetDoc** using dot notation, to access documents that are "nested" within other documents. For example, the following accesses a document nested three levels deep:

```
&Docs3 = &CalcCostOut.GetDoc("Doc1.Doc2.Doc3");
```

Parameters

Parameter	Description
docname	The name of the document that **GetDoc** should get from the output structure.

Returns

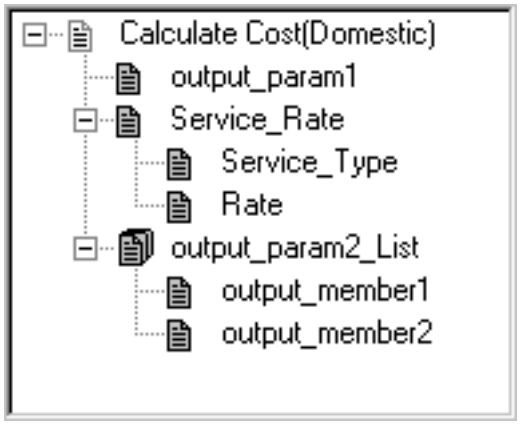
A reference to the document received from the output structure.

Example

In the following example, the Output structure for Calculate Cost(Domestic), or the root level document, is created with the **GetOutputDocs** method. (To create, or get, more Output structures, call **GetNextDoc**.) The Calculate Cost output parameter Service_Rate is a document, so the **GetDoc** method gets it from the Output structure.

```
Local Interlink &QE_FEDEX_COST;
Local number &count;
Local BIDocs &CalcCostOut;
Local BIDocs &ServiceRateDoc;
Local number &ret;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);
// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");
&ret = &CalcCostOut.GetValue("output_param1",&VALUE);
&ServiceRateDoc = &CalcCostOut.GetDoc("Service_Rate");
&ret = &ServiceRateDoc.GetValue("Service_Type", &SERVICE_TYPE);
&ret = &ServiceRateDoc.GetValue("Rate", &RATE);
/* Call GetDoc, GetValue, GetNextDoc for output_param2_List (code not shown) */
If &ret = 0 Then
   /* Process output values */
End-If;
```

The following illustration shows the Output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List. This is a version of the Federal Express plug-in that was modified for this example (output_param1 and output_param2_List were added).



Example output structure

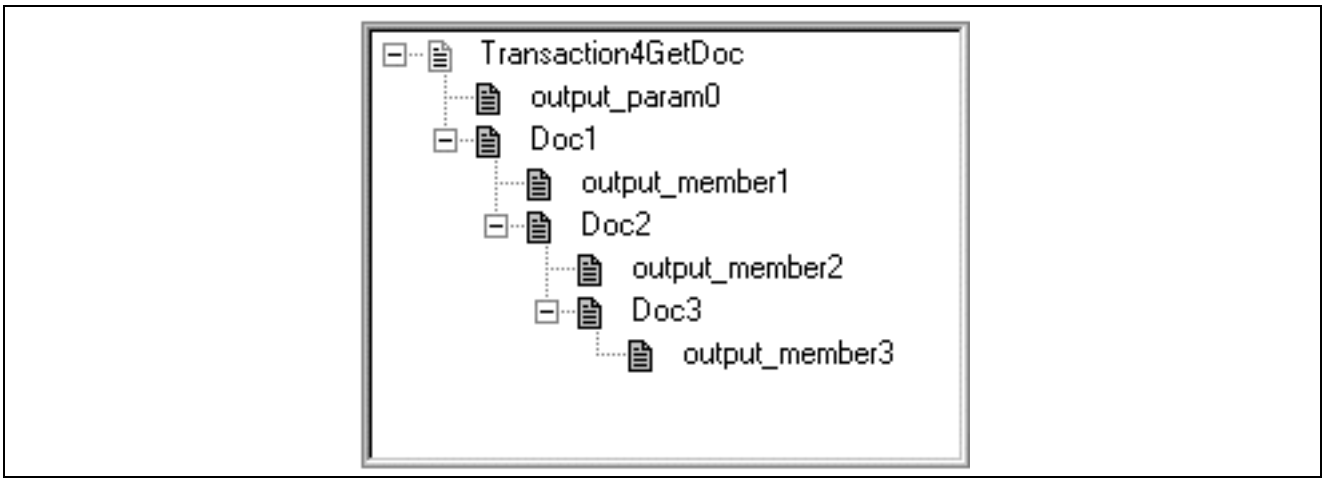
In the following example, **GetDoc** is used to access a document that is nested more deeply. To access a document that is more deeply nested, you can either call **GetDoc** for each nesting, or you can call **GetDoc** once using the nesting feature.

Calling **GetDoc** with the nesting feature:

```
Local Interlink &QE_4GETDOC;
Local BIDocs &CalcCostOut, &Docs3;
Local number &ret;

&QE_4GETDOC = GetInterlink(Interlink.QE_4GETDOC_EX);
// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_4GETDOC.GetOutputDocs("");
&Docs3 = &CalcCostOut.GetDoc("Doc1.Doc2.Doc3");
&ret = &Docs3.GetValue("output_member3", &VALUE);
```

Calling **GetDoc** without the nesting feature:

```
Local Interlink &QE_4GETDOC;
Local BIDocs &CalcCostOut, &Docs1, &Docs2, &Docs3;
Local number &ret;

&QE_4GETDOC = GetInterlink(Interlink.QE_4GETDOC_EX);
// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_4GETDOC.GetOutputDocs("");
&Docs1 = &CalcCostOut.GetDoc("Doc1");
&Docs2 = &Docs1.GetDoc("Doc2");
&Docs3 = &Docs2.GetDoc("Doc3");
&ret = &Docs3.GetValue("output_member3", &VALUE);
```



Example output structure with nested documents

GetFieldCount

Syntax

GetFieldCount()

Description

Use the **GetFieldCount** method to support dynamic output. It returns the number of columns in the output buffer, which is the same as the number of outputs in each row of the output buffer.

Note. This method can also be used with hierarchical doc data.

Parameters

None.

Returns

The number of columns in the output buffer, which is the same as the number of outputs in each row of the output buffer.

Example

The following example moves to the first column, first field of the output buffer. The **Repeat** loop goes through every row in the output buffer. The **For** loop processes every field in every row.

```
If (&MYBI.MoveFirst()) Then
   Repeat
      For &I = 1 to &MYBI.GetFieldCount
         &TYPE = &MYBI.GetFieldType(&I);
         Evaluate &TYPE
         Where = 1
            /* do processing */
         .  .  .
         End-Evaluate;
      End-For;
```

```
   Until Not (&MYBI.MoveNext());
Else
   /* do error processing – output buffer not returned */
End-If;
```

See Also

Chapter 8, Business Interlink Class , GetFieldType, page 692, Chapter 8, Business Interlink Class , GetFieldValue, page 693, Chapter 8, Business Interlink Class , MoveFirst, page 704, Chapter 8, Business Interlink Class , MoveNext, page 705, Chapter 8, Business Interlink Class , Using Hierarchical Data (BIDocs), page 662.

GetFieldType

Syntax

GetFieldType(*index*)

Description

Use the **GetFieldType** method to support dynamic output. It returns the type of field specified by *index*. You can use this method only after you have used the **MoveFirst** method: otherwise the system doesn't know where to start.

Note. This method can also be used with hierarchical doc data.

Parameter

index Specify the number of the field you want to find the type of.

Returns

A number indicating the type of the field. The values are:

Value	Description
1	String
2	Integer
3	Float
4	Boolean
5	Date
6	Time
7	DateTime
8	Binary
9	Object

Example

In the following example, the Business Interlink object name is &MYBI. The example uses **MoveFirst** to move to the first row of the output buffer, and to the first column, or first field, in that row. The **Repeat** loop uses **MoveNext** to go through every row in the output buffer. The **For** loop processes every field in every row, using the **GetFieldCount** method to get the number of fields, or outputs, in the row. Within the **For** loop, **GetFieldType** gets the type of the field data (string, integer, and so on.)

```
/* Add inputs to the Business Interlink object, then call Execute to execute the⇒
 Business Interlink object. You are then ready to get the outputs using the⇒
 following code. */

If (&MYBI.MoveFirst()) Then
   Repeat
      For &I = 1 to &MYBI.GetFieldCount
         &TYPE = &MYBI.GetFieldType(&I);
         Evaluate &TYPE
         Where = 1
            &STRING_VARIABLE = &MYBI.GetFieldValue(&I);
         /* test for and process other field types */
         End-Evaluate;
      End-For;
   Until Not(&MYBI.MoveNext());
Else
   /* Process error − no output buffer */
End-If;
```

See Also

Chapter 8, Business Interlink Class , GetFieldCount, page 691, Chapter 8, Business Interlink Class , GetFieldValue, page 693, Chapter 8, Business Interlink Class , MoveFirst, page 704, Chapter 8, Business Interlink Class , MoveNext, page 705.

GetFieldValue

Syntax

GetFieldValue(*index*)

Description

Use the **GetFieldValue** method to support dynamic output. It returns the value of the field specified by *index*. You can use this method only after you have used the **MoveFirst** method: otherwise the system doesn't know where to start.

Note. This method can also be used with hierarchical doc data.

Parameter

index Specify the number of the field you want to find the value of.

Returns

A string containing the value of the field.

Example

In the following example, the Business Interlink object name is &MYBI. The example uses **MoveFirst** to move to the first row of the output buffer, and to the first column, or first field, in that row. The **Repeat** loop uses **MoveNext** to go through every row in the output buffer. The **For** loop processes every field in every row, using the **GetFieldCount** method to get the number of fields, or outputs, in the row. Within the **For** loop, **GetFieldValue** gets the value of the field data.

```
/* Add inputs to the Business Interlink object, then call Execute to execute the⇒
 Business Interlink object. You are then ready to get the outputs using the⇒
 following code. */

If (&MYBI.MoveFirst()) Then
   Repeat
      For &I = 1 to &MYBI.GetFieldCount
         &TYPE = &MYBI.GetFieldType(&I);
         Evaluate &TYPE
         Where = 1
            &STRING_VARIABLE = &MYBI.GetFieldValue(&I);
         /* test for and process other field types */
         End-Evaluate;
      End-For;
   Until Not(&MYBI.MoveNext());
Else
   /* Process error − no output buffer */
End-If;
```

See Also

Chapter 8, Business Interlink Class , GetFieldCount, page 691, Chapter 8, Business Interlink Class , GetFieldType, page 692, Chapter 8, Business Interlink Class , MoveFirst, page 704, Chapter 8, Business Interlink Class , MoveNext, page 705.

GetInputDocs

Syntax

```
GetInputDocs("")
```

Description

The **GetInputDocs** method gets the top input document at the root level for a Business Interlink object. This is a hierarchical structure that contains the values for the inputs for this Business Interlink object. The methods that you use to put the input values into this document are the hierarchical data methods.

Parameters

A null string.

Returns

An input document. This is the document at the top of the root level of the input document for a Business Interlink object.

Example

```
Local Interlink &QE_FEDEX_COST;
Local BIDocs &CalcCostOut, &CalcCostIn;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);
&CalcCostIn = &QE_FEDEX_COST.GetInputDocs("");

/* You can now insert the input values and execute the Business Interlink object.⇒
 */
```

See Also

Chapter 8, Business Interlink Class , Using Hierarchical Data (BIDocs), page 662.

GetNextDoc

Syntax

GetNextDoc()

Description

The **GetNextDoc** method gets a document from one of the following levels:

- The root level of the Output structure for a Business Interlink object. This level was accessed with the **GetOutputDocs** method.
- When a document within the Output structure is a list, **GetNextDoc** gets another document from the list. If you use **GetNextDoc** on a document that is not a list, **GetNextDoc** fails and returns an error.

Parameters

None.

Returns

Number: Enum value and Meaning

0: eNoError. The method succeeded.

1: eNO_DOCUMENT. The document referenced by this method does not exist.

2: eDOCLIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time results in this error.

3: eDOCUMENT_UNINITIALIZED. Internal error.

4: eNOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.

5: eNOT_LISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.

7: eNOT_BASICTYPE. You tried to perform a GetValue or AddValue upon a parameter that is not of basic type: integer, float, string, time, date, datetime.

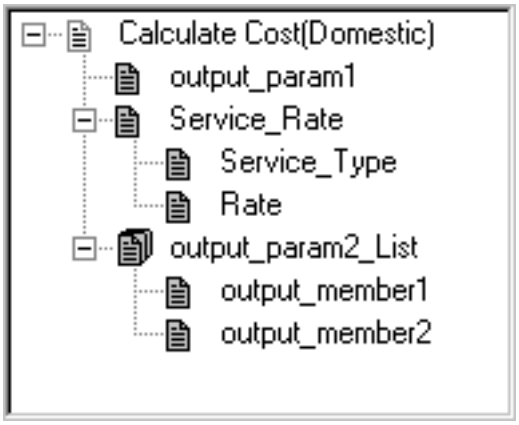
8: eNO_DATA. You tried to retrieve data from a document that contained no data.

Example

In this example, the Output structure for Calculate Cost, or the root level document, is accessed with the **GetOutputDocs** method. The **GetNextDoc** method gets another Output structure, assuming that there is more than one of them. The Calculate Cost output parameter output_param2_List is a document, so the **GetDoc** method gets it to the Output structure. output_param2_List is also a document list, so **GetNextDoc** method gets the next output_param2_List document.

```
Local Interlink &QE_FEDEX_COST;
Local number &count1, &count2;
Local BIDocs &CalcCostOut;
Local BIDocs &OutlistDoc;
Local number &ret1, &ret2;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);
// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

&ret1 = 0;
While (&ret1)
   &ret1 = &CalcCostOut.GetValue("output_param1",&VALUE);
   &ServiceRateDoc = &CalcCostOut.GetDoc("Service_Rate");
   &ret1 = &ServiceRateDoc.GetValue("Service_Type", &SERVICE_TYPE);
   &ret1 = &ServiceRateDoc.GetValue("Rate", &RATE);
   /* Process output values */

   &OutlistDoc = &CalcCostOut.GetDoc("output_param2_List");
   &count2 = &CalcCostOut.GetCount("output_param2_List");
   While (&I < &count2)
      &ret2 = &OutlistDoc.GetValue("output_member1", &VALUE1);
      &ret2 = &OutlistDoc.GetValue("output_member2", &VALUE2);
      If &ret2 = 0 Then
         /* Process output values */
         &I = &I + 1;
         &ret2 = &OutlistDoc.GetNextDoc();
      End-If;
   &ret1 = &CalcCostOut.GetNextDoc();
End-While;
```

The following shows the Output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List.



Example output structure

GetOutputDocs

Syntax

GetOutputDocs("")

Description

The **GetOutputDocs** method gets the top output document at the root level for a Business Interlink object. This is a hierarchical structure that contains the values for the outputs for this Business Interlink object. The methods that you use to get output values from this document are the hierarchical data methods.

Parameters

A null string.

Returns

An output document. This is the document at the top of the root level of the output document for a Business Interlink object.

Example

```
Local Interlink &QE_FEDEX_COST;
Local BIDocs &CalcCostIn, &CalcCostOut;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

/* You can now execute the Business Interlink object and get the output values. */
```

See Also

Chapter 8, Business Interlink Class , Using Hierarchical Data (BIDocs), page 662.

GetPreviousDoc

Syntax

```
GetPreviousDoc()
```

Description

The **GetPreviousDoc** method gets the previous document from either the root level of the Output structure for a Business Interlink object, or from a document within the Output structure. When these documents are in a list, **GetPreviousDoc** gets the previous document in the list. You must get the document before you can get its values with **GetValue**.

Parameters

None.

Returns

Number: Enum value and Meaning

0: eNoError. The method succeeded.

1: eNO_DOCUMENT. The document referenced by this method does not exist.

2: eDOCLIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time results in this error.

3: eDOCUMENT_UNINITIALIZED. Internal error.

4: eNOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.

5: eNOT_LISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.

7: eNOT_BASICTYPE. You tried to perform a GetValue or AddValue upon a parameter that is not of basic type: integer, float, string, time, date, datetime.

8: eNO_DATA. You tried to retrieve data from a document that contained no data.

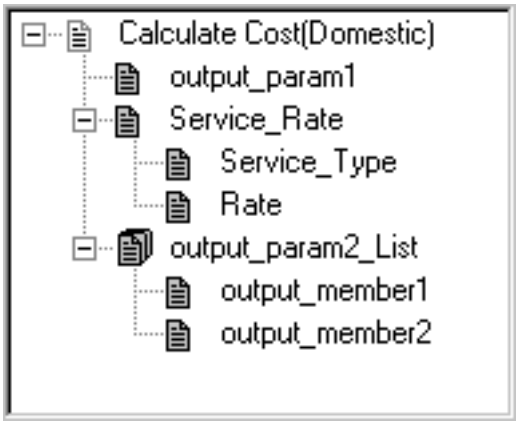
Example

In this example, the Output structure for Calculate Cost, or the root level document, is accessed with the **GetOutputDocs** method. It contains one output parameter, output_param2_List, which is also a list. The **GetCount** and **MoveToDoc** methods point to the last output_param2_List document in the list. The **GetPreviousDoc** method is used in a loop to cycle through the output_param2_List list, starting with the last and ending with the first in the list, and get each output_param2_List document and its values.

```
Local Interlink &QE_FEDEX_COST;
Local number &count;
Local BIDocs &CalcCostOut;
Local BIDocs &OutlistDoc;
Local number &ret;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);
// Get inputs, execute. (code not shown)
```

```
&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

/* Call GetValue for output_param1, call GetDoc, GetValue for Service_Rate (code⇒
 not shown) */

&OutlistDoc = &CalcCostOut.GetDoc("output_param2_List");
&count = &CalcCostOut.GetCount("output_param2_List");
&ret = &OutlistDoc.MoveToDoc(&count-1);
&I = &count;
While (&I > 0)
   &ret = &OutlistDoc.GetValue("output_member1", &VALUE1);
   &ret = &OutlistDoc.GetValue("output_member2", &VALUE2);
   If &ret = 0 Then
      /* Process output values */
      &I = &I - 1;
      &retoutput = &OutlistDoc.GetPreviousDoc("");
   End-If;
End-While;
```

The following shows the Output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List.



Example output structure

GetStatus

Syntax

GetStatus()

Description

The **GetStatus** method tests the BIDocs document. To find the status for any BIDocs method that does not return a status value, call GetStatus just after you call that BIDocs method.

Parameters

None.

Returns

Number: Enum value and Meaning

0: NoError. The method succeeded.

1: NO_DOCUMENT. The document referenced by this method does not exist.

2: LIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time results in this error.

3: DOCUMENT_UNINITIALIZED. Internal error.

4: NOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.

5: NOT_DOCUMENTLISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.

6: NOT_LISTTYPE. You tried to perform a list operation using GetValue, AddValue, on a non-list.

7: NOT_SINGLEBASICTYPE. You tried to perform a GetValue or AddValue upon a list that does not use a single basic type: integer, float, string, time, date, datetime.

9: NO_DATA. You tried to retrieve data from a document that contained no data.

10: GENERIC_ERROR. There was an error with the transaction.

GetValue

Syntax

GetValue(*paramname, value*)

Description

The **GetValue** method gets a value from an output parameter within a document of the Output structure for a Business Interlink object.

Parameters

Parameter	Description
Paramname	The name of the member of the document that is having a value retrieved from it.
Value	The value that is retrieved. This can be a variable or a record field. The data type can be string, integer, double, float, time, date, or datetime.

Returns

Value: Meaning

string, *value* : The value of the output parameter.

Integer: When the syntax is integer GetValue(*name*, *value*), integer is the return status of GetValue, listed in the following table.

Return Status for Integer

Number	Enum value and Meaning
0	NoError. The method succeeded.
1	NO_DOCUMENT. The document referenced by this method does not exist.
9	NO_DATA. You tried to retrieve data from a document that contained no data.
10	GENERIC_ERROR. There was an error with the transaction.

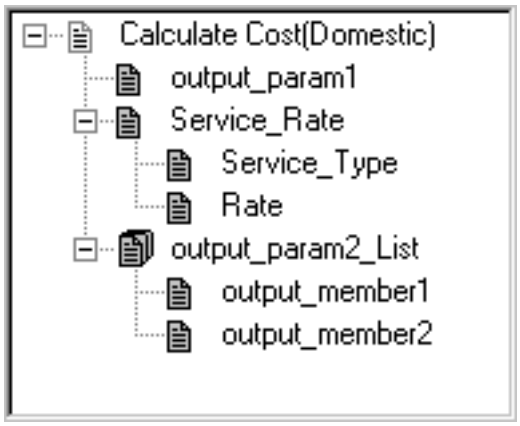
Example

In the following example, the Business Interlink object name is QE_FEDEX_COST.

In the following example, the Output structure for Calculate Cost, or the root level document, is created with the **GetOutputDocs** method. (If you wanted to create, or get, more Output structures, you would call **GetNextDoc**.) The Calculate Cost output parameter Service_Rate is a document, so the **GetDoc** method gets it from the Output structure. Then the **GetValue** method gets values from each of the Service_Rate document members.

```
Local Interlink &QE_FEDEX_COST;
Local BIDocs &CalcCostOut;
Local BIDocs &ServiceRateDoc;
Local number &ret;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");
&ret = &CalcCostOut.GetValue("output_param1",&PARAM1);

&ServiceRateDoc = &CalcCostOut.GetDoc("Service_Rate");
&ret = &ServiceRateDoc.GetValue("Service_Type", &SERVICE_TYPE);
&ret = &ServiceRateDoc.GetValue("Rate", &RATE);

/* Call GetDoc, GetValue, GetNextDoc for output_param2_List (code not shown) */
```

The following illustration shows the Output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List. This is a version of the Federal Express plug-in that was modified for this example (output_param1 and output_param2_List were added).



Example output structure

InputRowset

Syntax

InputRowset(*&Rowset*)

Description

The **InputRowset** method uses the data in the specified rowset to populate the input buffer, copying *like-named* fields in the appropriate structure. This method assumes that the names of the fields in the rowset match the names of the inputs defined in the Business Interlink definition, *and* that the structure of both rowsets are the same.

Use this method only if you have a hierarchical data structure and you want to preserve the hierarchy. Otherwise, use **BulkExecute** or **AddInputRow**.

Parameters

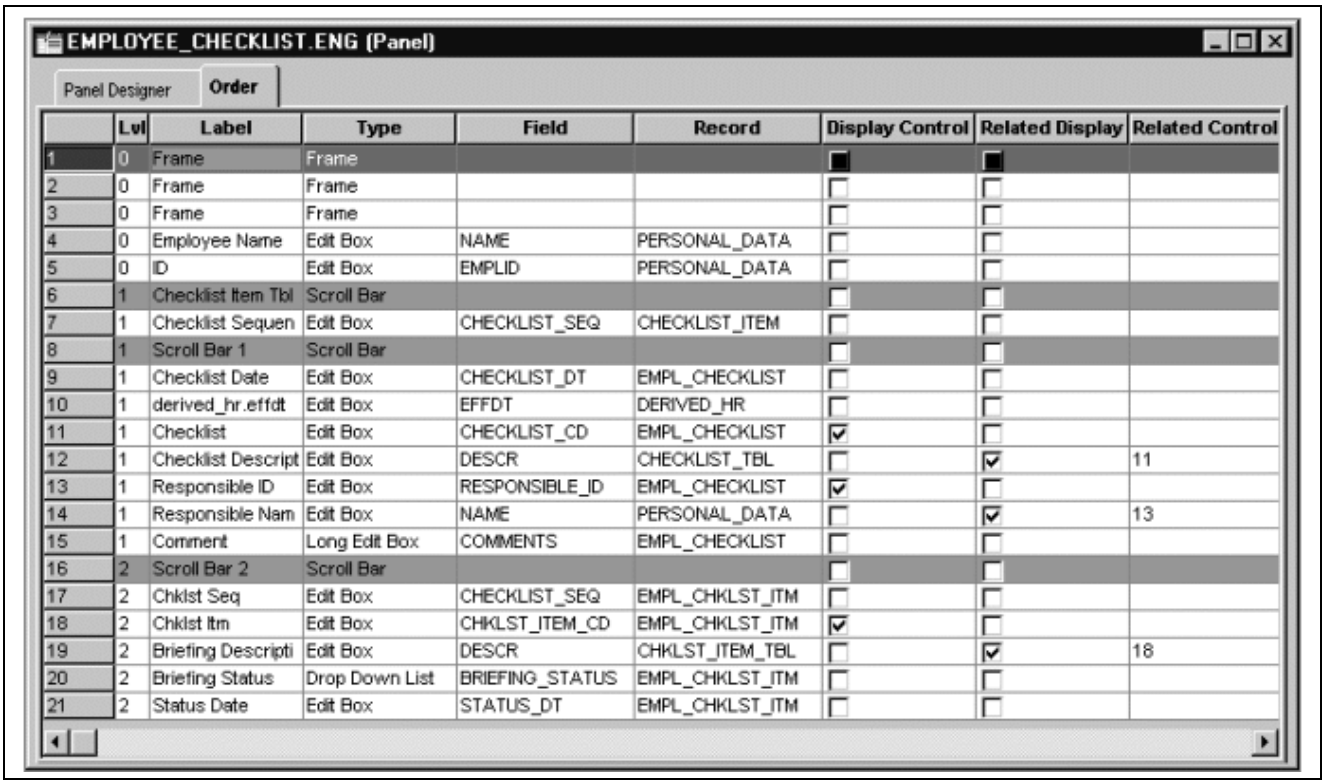
Parameter	Description
&Rowset	Specify an existing, instantiated rowset object.

Returns

An optional value: the number of rows inserted into the output buffer.

Example

The example uses the rowset on level 1 from the EMPLOYEE_CHECKLIST page. A PeopleCode program running in a field on level 0 in that panel can access the child rowset (level 1), shown below from Scroll Bar 1 to Scroll Bar 2.



	Lvl	Label	Type	Field	Record	Display Control	Related Display	Related Control
1	0	Frame	Frame			■	■	
2	0	Frame	Frame			☐	☐	
3	0	Frame	Frame			☐	☐	
4	0	Employee Name	Edit Box	NAME	PERSONAL_DATA	☐	☐	
5	0	ID	Edit Box	EMPLID	PERSONAL_DATA	☐	☐	
6	1	Checklist Item Tbl	Scroll Bar			☐	☐	
7	1	Checklist Sequen	Edit Box	CHECKLIST_SEQ	CHECKLIST_ITEM	☐	☐	
8	1	Scroll Bar 1	Scroll Bar			☐	☐	
9	1	Checklist Date	Edit Box	CHECKLIST_DT	EMPL_CHECKLIST	☐	☐	
10	1	derived_hr.effdt	Edit Box	EFFDT	DERIVED_HR	☐	☐	
11	1	Checklist	Edit Box	CHECKLIST_CD	EMPL_CHECKLIST	☑	☐	
12	1	Checklist Descript	Edit Box	DESCR	CHECKLIST_TBL	☐	☑	11
13	1	Responsible ID	Edit Box	RESPONSIBLE_ID	EMPL_CHECKLIST	☑	☐	
14	1	Responsible Nam	Edit Box	NAME	PERSONAL_DATA	☐	☑	13
15	1	Comment	Long Edit Box	COMMENTS	EMPL_CHECKLIST	☐	☐	
16	2	Scroll Bar 2	Scroll Bar			☐	☐	
17	2	Chklst Seq	Edit Box	CHECKLIST_SEQ	EMPL_CHKLST_ITM	☐	☐	
18	2	Chklst Itm	Edit Box	CHKLST_ITEM_CD	EMPL_CHKLST_ITM	☑	☐	
19	2	Briefing Descripti	Edit Box	DESCR	CHKLST_ITEM_TBL	☐	☑	18
20	2	Briefing Status	Drop Down List	BRIEFING_STATUS	EMPL_CHKLST_ITM	☐	☐	
21	2	Status Date	Edit Box	STATUS_DT	EMPL_CHKLST_ITM	☐	☐	

EMPLOYEE_CHECKLIST Page structure

EMPL_CHECKLIST is the primary database record for the level 1 scrollbar on the EMPLOYEE_CHECKLIST page. The following PeopleCode access the level 1 rowset using EMPL_CHECKLIST.
The Business Interlink object name is QE_BI_EMPL_CHECKLIST. This Business Interlink object uses the level 1 rowset as its input and its output.

The **InputRowset** method uses this rowset as input for QE_BI_EMPL_CHECKLIST. Then a blank duplicate of the rowset is created with CreateRowset, and then the output of QE_BI_EMPL_CHECKLIST is fetched into the blank rowset with **FetchIntoRowset**.

```
&MYROWSET = GetRowset(SCROLL.EMPL_CHECKLIST);
&ROWCOUNT = &QE_BI_EMPL_CHECKLIST.InputRowset(&MYROWSET);
&RSLT = &QE_BI_EMPL_CHECKLIST.Execute();
/* do some error processing */
&WorkRowset = CreateRowset(&MYROWSET);
&ROWCOUNT = &QE_BI_EMPL_CHECKLIST.FetchIntoRowset(&WorkRowset);

If &ROWCOUNT = 0 Then
/* do some error processing */
Else
   /* Process the rowset from QE_BI_EMPL_CHECKLIST. Check it for errors. */
   For &I = 1 to &WorkRowset.RowCount
      For &K = 1 to &WorkRowset(&I).RecordCount
         &REC = &WorkRowset(&I).GetRecord(&K);
         &REC.ExecuteEdits();
         For &M = 1 to &REC.FieldCount
            If &REC.GetField(&M).EditError Then
            /* there are errors */
```

```
            /* do other processing */
            End-If;
         End-For;
      End-For;
   End-for;
End-if;
```

See Also

Chapter 8, Business Interlink Class , FetchIntoRowset, page 683, *PeopleCode Developer's Guide,* Data Buffer Access.

MoveFirst

Syntax

MoveFirst()

Description

Use the **MoveFirst** method to support dynamic output. This method moves your cursor to the first column, first row of the output buffer. You must use this method *before* you try to access any data.

Parameters

None.

Returns

Boolean: True if successfully positioned cursor at the first column, first row of the output buffer, False otherwise.

Example

In the following example, the Business Interlink object name is &MYBI. The example uses **MoveFirst** to move to the first row of the output buffer, and to the first column, or first field, in that row. The **Repeat** loop uses **MoveNext** to go through every row in the output buffer. The **For** loop processes every field in every row, using the **GetFieldCount** method to get the number of fields, or outputs, in the row.

```
/* Add inputs to the Business Interlink object, then call Execute to execute the⇒
 Business Interlink object. You are then ready to get the outputs using the⇒
 following code. */

If (&MYBI.MoveFirst()) Then
   Repeat
      For &I = 1 to &MYBI.GetFieldCount
         &TYPE = &MYBI.GetFieldType(&I);
         Evaluate &TYPE
         Where = 1
            &STRING_VARIABLE = &MYBI.GetFieldValue(&I);
         /* test for and process other field types */
         End-Evaluate;
      End-For;
   Until Not(&MYBI.MoveNext());
Else
```

```
   /* Process error – no output buffer */
End-If;
```

See Also

Chapter 8, Business Interlink Class , GetFieldCount, page 691, Chapter 8, Business Interlink Class , GetFieldType, page 692, Chapter 8, Business Interlink Class , GetFieldValue, page 693, Chapter 8, Business Interlink Class , MoveNext, page 705.

MoveNext

Syntax

MoveNext()

Description

Use the **MoveNext** method to support dynamic output. This method moves your cursor to the first column of the next row of the output buffer. You can use this method only after you have used the **MoveFirst** method: otherwise, the system doesn't know where to start.

Parameters

None.

Returns

Boolean: True if successfully positioned cursor at the next row of the output buffer, False otherwise.

Example

In the following example, the Business Interlink object name is &MYBI. The example uses **MoveFirst** to move to the first row of the output buffer, and to the first column, or first field, in that row. The **Repeat** loop uses **MoveNext** to go through every row in the output buffer. The **For** loop processes every field in every row, using the **GetFieldCount** method to get the number of fields, or outputs, in the row.

```
/* Add inputs to the Business Interlink object, then call Execute to execute the⇒
 Business Interlink object. You are then ready to get the outputs using the⇒
 following code. */

If (&MYBI.MoveFirst()) Then
   Repeat
      For &I = 1 to &MYBI.GetFieldCount
         &TYPE = &MYBI.GetFieldType(&I);
         Evaluate &TYPE
         Where = 1
            &STRING_VARIABLE = &MYBI.GetFieldValue(&I);
         /* test for and process other field types */
         End-Evaluate;
      End-For;
   Until Not(&MYBI.MoveNext());
Else
   /* Process error – no output buffer */
End-If;
```

See Also

Chapter 8, Business Interlink Class , GetFieldCount, page 691, Chapter 8, Business Interlink Class , GetFieldType, page 692, Chapter 8, Business Interlink Class , GetFieldValue, page 693, Chapter 8, Business Interlink Class , MoveFirst, page 704.

MoveToDoc

Syntax

MoveToDoc(*list_number*)

Description

Within a list of documents in the Output structure, the **MoveToDoc** method moves to the documents given by the parameter *list_number*. After using **MoveToDoc**, the **GetValue** method gets the values of the document that is in the *list_number*+1 location in the list.

Parameters

Parameter	Description
list_number	The number indicating the document that **MoveToDoc** moves to. After using **MoveToDoc**, the **GetValue** method gets the values of the document that is in the *list_number*+1 location in the list. For example, if *list_number* is zero, then **MoveToDoc** moves to the first document in the list.

Returns

Number: Enum value and Meaning

0: eNoError. The method succeeded.

1: eNO_DOCUMENT. The document referenced by this method does not exist.

2: eDOCLIST_OUT_RANGE. You tried to access a document in a list, but the document list is out of range. For example, if a document list contains five documents, and then you call GetDoc/GetOutputDocs once, you can call GetNextDoc four times; the fifth time results in this error.

3: eDOCUMENT_UNINITIALIZED. Internal error.

4: eNOT_DOCUMENTTYPE. You tried to perform an operation upon a parameter that is not a document type.

5: eNOT_LISTTYPE. You tried to perform a GetNextDoc or AddNextDoc upon a document that is not a list.

7: eNOT_BASICTYPE. You tried to perform a GetValue or AddValue upon a parameter that is not of basic type: integer, float, string, time, date, datetime.

8: eNO_DATA. You tried to retrieve data from a document that contained no data.

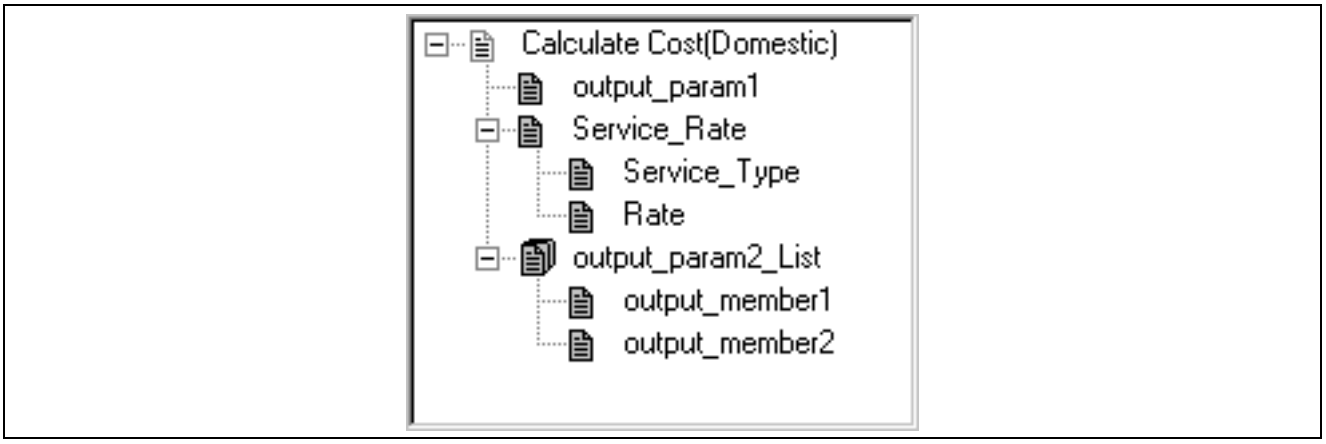
Example

The following example gets the values of the last document in the output_param2_List list. It uses **GetCount** to get the number of documents in the list, and then uses **MoveToDoc** to move to the last document in the list.

```
Local Interlink &QE_FEDEX_COST;
Local number &count;
```

```
Local BIDocs &CalcCostOut;
Local BIDocs &OutlistDoc;
Local number &ret;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);
// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");
&OutlistDoc = &CalcCostOut.GetDoc("output_param2_List");
&count = &CalcCostOut.GetCount("output_param2_List");
&ret = &OutlistDoc.MoveToDoc(&count-1);
&ret = &OutlistDoc.GetValue("output_member1", &VALUE1);
&ret = &OutlistDoc.GetValue("output_member2", &VALUE2);
```

The following illustration shows the Output structure for this example. It contains three output parameters: output_param1, Service_Rate, and output_param2_List. This is a version of the Federal Express plug-in that was modified for this example (output_param1 and output_param2_List were added).



Example output structure

ResetCursor

Syntax

ResetCursor()

Description

The **ResetCursor** method resets the cursor in the Output structure for a Business Interlink object to the top. After you call this method, the next time you call **GetValue**, you get an output value from the first document of the Output structures for a Business Interlink object.

Parameters

None.

Returns

None.

Example

The following code uses **ResetCursor** to reset the cursor in the Output structure to the top.

```
Local Interlink &QE_FEDEX_COST;
Local BIDocs &CalcCostOut;

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

// Get inputs, execute. (code not shown)

&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

// Perform actions on the output documents (code not shown)

&CalcCostOut.ResetCursor();
```

Incoming Business Interlink Methods

This section describes the PeopleCode methods you use with Incoming Business Interlinks.

AddAttribute

Syntax

AddAttribute(*attributename*, *attributevalue*)

Description

The **AddAttribute** method adds an attribute name and its value to an XML element referenced by a BiDocs object.

Parameters

Parameter	Description
attributename	String. The name of the attribute.
attributevalue	String. The value of the attribute.

Returns

Number. The return status. NoError, or 0, means the method succeeded.

Example

Here is a set of XML response code.

```
<?xml version="1.0"?>
   <postreqresponse>
      <candidate>
```

```
            <user>
               <location scenery="great" density="low" blank="eh?">
               </location>
            </user>
         </candidate>
      </postreqresponse>
```

Here is the PeopleCode that builds it.

```
Local BIDocs &rootDoc, &postreqresponse;
Local BIDocs &candidates, &location, &user;
Local number &ret;

&rootDoc = GetBiDoc("");
&ret = &rootDoc.AddProcessInstruction("<?xml version=""1.0""?>");
&postreqresponse = &rootDoc.CreateElement("postreqresponse");
&candidates = &postreqresponse.CreateElement("candidates");
&user = &candidates.CreateElement("user");
&location = &user.CreateElement("location");
&ret = &location.AddAttribute("scenery", "great");
&ret = &location.AddAttribute("density", "low");
&ret = &location.AddAttribute("blank", "eh?");
```

See Also

Chapter 8, Business Interlink Class , GetStatus, page 699.

AddComment

Syntax

AddComment(*comment*)

Description

The **AddComment** method adds an XML comment after the beginning tag of an XML element referenced by a BiDoc object.

Parameters

Parameter	Description
comment	String. The comment.

Returns

Number. The return status. NoError, or 0, means the method succeeded.

Example

Here is a set of XML response code.

```
<?xml version="1.0"?>
   <postreqresponse>
```

```
      <error>
         <!--this is a comment line-->
         <errorcode>1</errorcode>
         <errortext></errortext>
      </error>
   </postreqresponse>
```

Here is the PeopeCode that builds it.

```
Local BIDocs &rootDoc, &postreqresponse, &error, &errorcode, &errortext;
Local string &blob;
Local number &ret;
&rootDoc = GetBiDoc("");

/* add a processing instruction*/
&ret = &rootDoc.AddProcessInstruction("<?xml version=""1.0""?>");
/* create an element and add text*/
&postreqresponse = &rootDoc.CreateElement("postreqresponse");
&error = &postreqresponse.CreateElement("error");
&ret = &error.AddComment("this is a comment line");
&errorcode = &error.CreateElement("errorcode");
&ret = &errorcode.AddText("1");
&errortext = &error.CreateElement("errortext");
```

See Also

Chapter 8, Business Interlink Class , GetStatus, page 699.

AddProcessInstruction

Syntax

AddProcessInstruction(*instruction*)

Description

The **AddProcessInstruction** method adds an XML processing instruction to a BiDocs object. Use this method at the root level of the BiDocs object for Incoming Business Interlinks before you add anything else to the BiDocs object.

Parameters

Parameter	Description
instruction	String. The processing instruction.

Returns

Number. The return status. NoError, or 0, means the method succeeded.

Example

Here is a set of XML response code.

```
<?xml version="1.0"?>
   <postreqresponse>
      <error>
         <!--this is a comment line-->
         <errorcode>1</errorcode>
         <errortext></errortext>
      </error>
   </postreqresponse>
```

Here is the PeopeCode that builds it.

```
Local BIDocs &rootDoc, &postreqresponse;
Local BIDocs &error, &errorcode, &errortext;
Local number &ret;
&rootDoc = GetBiDoc("");

/* add a processing instruction*/
&ret = &rootDoc.AddProcessInstruction("<?xml version=""1.0""?>");
/* create an element and add text*/
&postreqresponse = &rootDoc.CreateElement("postreqresponse");
&error = &postreqresponse.CreateElement("error");
&ret = &error.AddComment("this is a comment line");
&errorcode = &error.CreateElement("errorcode");
&ret = &errorcode.AddText("1");
&errortext = &error.CreateElement("errortext");
```

See Also

Chapter 8, Business Interlink Class , GetStatus, page 699.

AddText

Syntax

AddText(*text*)

Description

The **AddText** method adds text to an XML element referenced by a BiDocs object.

Parameters

Parameter	Description
text	String. The text.

Returns

Number. The return status. NoError, or 0, means the method succeeded.

Example

Here is a set of XML response code.

```
<?xml version="1.0"?>
   <postreqresponse>
      <error>
         <!--this is a comment line-->
         <errorcode>1</errorcode>
         <errortext></errortext>
      </error>
   </postreqresponse>
```

Here is the PeopeCode that builds it.

```
Local BIDocs &rootDoc, &postreqresponse;
Local BIDocs &error, &errorcode, &errortext;
Local number &ret;
&rootDoc = GetBiDoc("");

/* add a processing instruction*/
&ret = &rootDoc.AddProcessInstruction("<?xml version=""1.0""?>");
/* create an element and add text*/
&postreqresponse = &rootDoc.CreateElement("postreqresponse");
&error = &postreqresponse.CreateElement("error");
&ret = &error.AddComment("this is a comment line");
&errorcode = &error.CreateElement("errorcode");
&ret = &errorcode.AddText("1");
&errortext = &error.CreateElement("errortext");
```

See Also

Chapter 8, Business Interlink Class , GetStatus, page 699.

CreateElement

Syntax

CreateElement(*elementname*)

Description

The **CreateElement** method creates an XML element with the given name within a BiDoc object.

Parameters

Parameter	Description
elementname	String. The XML element name.

Returns

BiDocs. The reference to the created element.

Example

Here is a set of XML response code.

```
<?xml version="1.0"?>
   <postreqresponse>
      <error>
         <errorcode>1</errorcode>
         <errortext></errortext>
      </error>
   </postreqresponse>
```

Here is the PeopeCode that builds it.

```
Local BIDocs &rootDoc, &postreqresponse;
Local BIDocs &error, &errorcode, &errortext;
Local number &ret;

&rootDoc = GetBiDoc("");

/* add a processing instruction*/
&ret = &rootDoc.AddProcessInstruction("<?xml version=""1.0""?>");
/* create an element and add text*/
&postreqresponse = &rootDoc.CreateElement("postreqresponse");
&error = &postreqresponse.CreateElement("error");
&errorcode = &error.CreateElement("errorcode");
&ret = &errorcode.AddText("1");
&errortext = &error.CreateElement("errortext");
```

GenXMLString

Syntax

GenXMLString()

Description

The **GenXMLString** method creates an XML string from a BiDocs object. The BiDocs object must contain the shape and data needed for an XML string. This is part of the Incoming Business Interlink functionality, which enables PeopleCode to receive an XML request and return an XML response.

Parameters

None.

Returns

String. The XML string containing the shape and data of the BiDocs object. For example, you can use this method to create an XML string containing an XML response.

Example

The following example takes a BiDocs structure that contains an XML response and puts that into a text string. After this is done, the %Response.Write function can send this as an XML response.

```
Local BIDocs &rootDoc;
Local string &xmlString;
```

```
/* Create a BiDoc structure containing the data and shape of your XML response ⇒
(code not shown) */

/* Generate the XML string containing the data and shape of your XML response from⇒
 the BiDoc structure */
&xmlString = &rootDoc.GenXMLString();
%Response.Write(&xmlString);
```

GetAttributeName

Syntax

GetAttributeName(*attributenumber*)

Description

The **GetAttributeName** method gets the name of an attribute within an XML element referenced by a BiDocs object.

Parameters

Parameter	Description
attributenumber	Number. The index number of the attribute.

Returns

String. The name of the attribute.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>
   <postreq>
      <email>joe_blow@peoplesoft.com</email>
      <location scenery="great" density="low" blank="eh?">
         <city>San Rafael</city>
         <state>CA</state>
         <zip>94522</zip>
         <country>US</country>
      </location>
   </postreq>
```

Here is the PeopleCode that gets the name of the second attribute in the location node. &attrName is density.

```
Local BIDocs &rootInDoc, postreqDoc, &locationDoc;
Local string &blob, &attrName;
&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);
&postreqDoc = &rootInDoc.GetNode("postreq");
&locationDoc = &postreqDoc.GetNode("location");
&attrName = &locationDoc.GetAttributeName(2);
```

GetAttributeValue

Syntax

GetAttributeValue({*attributename* | *attributenumber*})

Description

The **GetAttributeValue** method gets the value of an attribute within an XML element referenced by a BiDocs object.

Parameters

Parameter	Description
attributenumber | attributename	Specify the attribute that you want to get the value for. You can specify either the attribute number (1 for the first attribute, 2 for the second, and so on) or the attribute name (an XML tag.)

Returns

String. The value of the attribute.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>
   <postreq>
      <email>joe_blow@peoplesoft.com</email>
      <location scenery="great" density="low" blank="eh?">
         <city>San Rafael</city>
         <state>CA</state>
         <zip>94522</zip>
         <country>US</country>
      </location>
   </postreq>
```

Here is the PeopleCode that gets the value of the second attribute in the location node. &attrValue is low.

```
Local BIDocs &rootInDoc, &postreqDoc, &locationDoc;
Local string &blob, &attrValue;
&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);
&postreqDoc = &rootInDoc.GetNode("postreq");
&locationDoc = &postreqDoc.GetNode("location");
&attrValue = &locationDoc.GetAttributeValue(2);
```

GetNode

Syntax

GetNode({*nodename* | *nodenumber*})

Description

The **GetNode** method returns a BiDocs reference to a child XML node (element or comment). Use the **GetNode** method upon a BiDocs reference to an XML element to access the child nodes for that element.

Parameters

Parameter	Description
nodenumber | nodename	Specify the node that you want to reference. You can specify either a node number (the first node is 1, the second 2, and so on) or a node name (that is, the XML tag.)

Returns

BiDocs. The returned XML element within a BiDocs object.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>
   <postreq>
      <email>joe_blow@peoplesoft.com</email>
      <projtitle>
         <projsubtitle>first_subtitle</projsubtitle>
         <projsubtitle>second_subtitle</projsubtitle>
         <projsubtitle>third_subtitle</projsubtitle>
      </projtitle>
   </postreq>
```

Here is the PeopleCode that gets the postreqDoc element and the projtitle element.

```
Local BIDocs &rootInDoc, &postreqDoc, &projtitleDoc;
Local string &name, &blob;
&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);
&postreqDoc = &rootInDoc.GetNode("postreq");
&projtitleDoc = &postreqDoc.GetNode("projtitle");
```

Here is the PeopleCode that gets the postreqDoc element, the projtitle element, and the third projsubtitle element.

```
Local BIDocs &rootInDoc, &postreqDoc, &projtitleDoc, &projsubtitleDoc;
Local string &name, &blob;
&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);
&postreqDoc = &rootInDoc.GetNode(1);
&projtitleDoc = &postreqDoc.GetNode(2);
&projsubtitleDoc = &projtitleDoc.GetNode(3);
```

To parse a list of elenents like <projsubtitle>, where elements have the same name, you must call **GetNode** using an index number rather than the element name.

ParseXMLString

Syntax

ParseXMLString(*XMLstring*)

Description

The **ParseXMLString** methods fills an existing BiDocs object with the data and shape from an XML string. This is part of the Incoming Business Interlink functionality, which enables PeopleCode to receive an XML request and return an XML response.

Parameters

Parameter	Description
XMLstring	A string containing an XML document.

Returns

Number. The return status. NoError, or 0, means the method succeeded.

Example

The following example gets an XML request, creates an empty BiDoc, and then fills the BiDoc with the data and shape contained in the XML string. After this is done, the GetDoc method and the GetValue method can get the value of the skills XML element, which is contained within the postreq element in the XML request.

```
Local BIDocs &rootInDoc, &postreqDoc;
Local string &blob;
Local number &ret;

&blob = %Request.GetContentBody();
/* process the incoming xml(request)- Create a BiDoc and fill with the request*/
&rootInDoc = GetBiDoc("");
&ret = &rootInDoc.ParseXMLString(&blob);
&postreqDoc = &rootInDoc.GetDoc("postreq");
&ret = &postreqDoc.GetValue("skills", &skills);
```

See Also

Chapter 8, Business Interlink Class , GetStatus, page 699.

Incoming Business Interlink Properties

This section describes the PeopleCode properties you use with Incoming Business Interlinks.

AttributeCount

Description

The **AttributeCount** property gets the number of attributes within an XML element referenced by a BiDocs object.

This property is read-only.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>
   <postreq>
      <email>joe_blow@peoplesoft.com</email>
      <location scenery="great" density="low" blank="eh?">
         <city>San Rafael</city>
         <state>CA</state>
         <zip>94522</zip>
         <country>US</country>
      </location>
   </postreq>
```

Here is the PeopleCode that gets the number of attributes in the location XML element. &count should be 3, for scenery, density, and blank.

```
Local BIDocs &rootInDoc, &postreqDoc, &locationDoc;
Local string &blob;
Local number &count;
&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);
&postreqDoc = &rootInDoc.GetNode("postreq");
&locationDoc = &postreqDoc.GetNode("location");
&count = &locationDoc.AttributeCount;
```

ChildNodeCount

Description

The **ChildNodeCount** property returns the number of XML child nodes within the element referenced by the BiDocs object. Child nodes include XML elements, comments, and processing instructions.

This property is read-only.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>
   <postreq>
      <email>joe_blow@peoplesoft.com</email>
      <projtitle>
```

```
         <!--this is a comment line-->
         <projsubtitle>first_subtitle</projsubtitle>
         <projsubtitle>second_subtitle</projsubtitle>
         <projsubtitle>third_subtitle</projsubtitle>
      </projtitle>
   </postreq>
```

Here is the XML code that gets the number of nodes within <postreq> and <projtitle>. &count1 is 2, for <email> and <projtitle>, and &count2 is 4, for the three <projsubtitle> nodes and the comment node.

```
Local BIDocs &rootInDoc, &projtitleDoc;
Local string &blob;
Local number &count1, &count2;
&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);
&postreqDoc = &rootInDoc.GetNode("postreq");
&count1 = &postreqDoc.ChildNodeCount;
&projtitleDoc = &postreqDoc.GetNode("projtitle");
&count2 = &projtitleDoc.ChildNodeCount;
```

NodeName

Description

The **NodeName** property gets the name of an XML element referenced by a BiDocs object. Use this to get the name of an XML element when you used GetNode with an index number to retrieve it (meaning that you did not have the name of the XML element when you used GetNode).

This property is read-only.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>
   <postreq>
      <email>joe_blow@peoplesoft.com</email>
      <projtitle>
         <projsubtitle>first_subtitle</projsubtitle>
         <projsubtitle>second_subtitle</projsubtitle>
         <projsubtitle>third_subtitle</projsubtitle>
      </projtitle>
   </postreq>
```

Here is the PeopleCode that gets the name of the <email> element, email.

```
Local BIDocs &rootInDoc, &postreqDoc, &emailDoc;
Local string &emailName, &blob;
&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);
&postreqDoc = &rootInDoc.GetNode(1);
```

```
&emailDoc = &postreqDoc.GetNode(1);
&emailName = &emailDoc.NodeName;
```

NodeType

Description

The **NodeType** property returns the type of an XML tag within a BiDocs object as an integer. The values are:

Value	Description
1	Element (a normal XML tag)
7	Processing instruction
8	Comment

This property is read-only.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>
   <postreq>
      <email>joe_blow@peoplesoft.com</email>
      <!--this is a comment-->
      <projtitle>
         <projsubtitle>first_subtitle</projsubtitle>
         <projsubtitle>second_subtitle</projsubtitle>
         <projsubtitle>third_subtitle</projsubtitle>
      </projtitle>
   </postreq>
```

Here is the PeopleCode that gets types: &xmlprocType is 7 for processing instruction, postreqDoc is 1 for element, and commentType is 8 for comment.

```
Local BIDocs &rootInDoc, &postreqDoc, &commentDoc;
Local number &xmlprocType, &postreqType, &commentDoc;
Local string &blob;
&blob = %Request.GetContentBody();

/* <?xml version="1.0"?> */
&rootInDoc = GetBiDoc(&blob);
&xmlprocType = &rootInDoc.NodeType;

/* <postreq> */
&postreqDoc = &rootInDoc.GetNode(1);
&postreqType = &postreqDoc.NodeType;

/* <!--this is a comment--> */
&commentDoc = &postreqDoc.GetNode(2);
&commentType = &commentDoc.NodeType;
```

NodeValue

Description

The **NodeValue** property returns the value of a node within an XML document as a string.

This property is read-only.

Example

Here is a set of XML request code.

```
<?xml version="1.0"?>
   <postreq>
      <email>joe_blow@peoplesoft.com</email>
      <projtitle>
         <projsubtitle>first_subtitle</projsubtitle>
         <projsubtitle>second_subtitle</projsubtitle>
         <projsubtitle>third_subtitle</projsubtitle>
      </projtitle>
   </postreq>
```

Here is the PeopleCode that gets the value of the third <projsubtitle> element, third_subtitle.

```
Local BIDocs &rootInDoc, &postreqDoc, &projtitleDoc, &projsubtitleDoc;
Local string &name, &blob;
&blob = %Request.GetContentBody();

&rootInDoc = GetBiDoc(&blob);
&postreqDoc = &rootInDoc.GetNode(1);
&projtitleDoc = &postreqDoc.GetNode(2);
&projsubtitleDoc = &projtitleDoc.GetNode(3);
&projsubtitleName = &projsubtitleDoc.NodeValue;
```

Business Interlink Class Property

This section explains the StopAtError property.

StopAtError

Description

The **StopAtError** property specifies whether the execution of the PeopleCode program stops when there's an error, or if the PeopleCode program tries to capture the errors.

This property takes a Boolean value: True to halt execution of your PeopleCode program at an error, False to continue executing. The default value is True.

This property is read-write.

Example

```
&QE_RP_SRAALL_1.StopAtError = False;
```

Configuration Parameters

There are two types of configuration parameters: the ones defined by the Interlink plug-in the Business Interlink definition is based on, and the ones that are standard, that is, defined for every Business Interlink object.

All configuration parameters must be set before you add any data to the input buffers.

In this section, we discuss the following parameters:

- Interlink plug-in configuration parameters
- URL configuration parameter
- BIDocValidate configuration parameter

Interlink Plug-in Configuration Parameters

The configuration parameters defined by the Interlink plug-in are accessed as if they were properties on the Business Interlink object. That is, in PeopleCode, you assign the value of a configuration parameter by using the Business Interlink object followed by a dot and the configuration parameter, in this format:

```
&MYINTERLINK.parameter = value;
```

You can also return the value of a configuration parameter:

```
&MYVALUE = &MYINTERLINK.parameter;
```

Each Business Interlink plug-in has its own set of configuration parameters. For example, the email project uses configuration parameters of SMTP_MAIL_SERVER, POP3MAIL_SERVER, LOGIN_NAME, PASSWORD, SENDERS_EMAIL_ADDRESS and REPLY_EMAIL_ADDRESS, while the Red Pepper transaction driver uses SERVER_NAME, RSERVER_HOST, RSERVER_PORT, and TIMEOUT.

You can specify default values for every configuration parameter in the Business Interlink definition (created in PeopleSoft Application Designer.) These values are used if you create a PeopleCode "template" by dragging the Business Interlink definition from the Project window in PeopleSoft Application Designer to an open PeopleCode editor window.

In the following example, the Interlink object name is QE_RP_SRAALL_1, and the driver is the Red Pepper driver:

```
&QE_RP_SRAALL_1.SERVER_NAME = "server";
&QE_RP_SRAALL_1.RSERVER_HOST = "pt-sun02.peoplesoft.com";
&QE_RP_SRAALL_1.RSERVER_PORT = "9884";
&QE_RP_SRAALL_1.TIMEOUT = 999;
&QE_RP_SRAALL_1.URL = "HTTP://www.PeopleSoft.com";
&QE_RP_SRAALL_1.StopAtError = False;
```

URL Configuration Parameter

Specifies the location and name of the Business Interlink plug-in to be used to connect to the external system. This configuration parameter takes a string value.

If the plug-in is located on a web server, you have to specify the name of the web server.

If you specify a file, you can specify either a relative or an absolute path:

- If you specify an absolute path, you must specify the drive letter:

```
&MYBI.URL = "File://D:\TEMP.MYDLL";
```

- If you specify a relative path, just use the file name:

```
&MYBI.URL = "File://TEMP.MYDLL";
```

If you specify a relative path, the system firsts looks for the file in the Location directory (specified by the user when the Business Interlink was first created), then it looks in the directory where PeopleTools is installed, in the PSTOOLS/Interface Drivers directory.

BIDocValidate Configuration Parameter

Specifies whether the system should verify whether the hierarchical data object (BIDoc) exists before adding or getting values from it. This configuration parameter takes a Boolean value: True if the system should verify before accessing the object, False otherwise.

The default value is True.

If this configuration parameter is specified as True, and the object specified doesn’t exist, the PeopleCode program halts execution and an error is displayed.

CHAPTER 9

Chart Class

This chapter provides an overview of the Peoplesoft Chart class and discusses the following topics:

- Chart terms.
- Chart types.
- Style sheets and charts.
- Component processor considerations.
- Translation considerations.
- iScripts and charts.
- Error handling.
- Data type of a chart object.
- Scope of a chart object.
- Chart class built-in functions.
- Chart class reference.

Understanding the Chart Class

The PeopleSoft Chart class enables you to plot data in charts according to your specifications. You can create a chart by setting data using PeopleCode. You can create a chart based on a chart control on a page, or at runtime using an Internet Script (iScript).

Possible uses for the chart control include:

- charting dynamic data, such as current prices of selected commodities, from a rowset.
- plotting stored data, such as product prices, for graphic analysis of trends, from a record.

The chart class supports the following features:

- Three-dimensional representation.
- Support for all major chart types.
- Control over the various elements of the chart such as the axes, legend, and titles.
- Interactive charts. The user can click on the different data elements of the chart to trigger further processing.

You can specify a rowset and have it graphed using a chart object with minimal PeopleCode. Within this rowset there must be a column containing data for the X axis and one for the Y axis. If more than one series of data is used an additional column should indicate the series values. Color and popup text for each point could also be set in this rowset using additional columns.

The following distinguishable types of data can be involved in a chart:

- Primary data.
- Secondary or overlay data.

The overlay data may be quite different to the primary data so an additional rowset is needed.

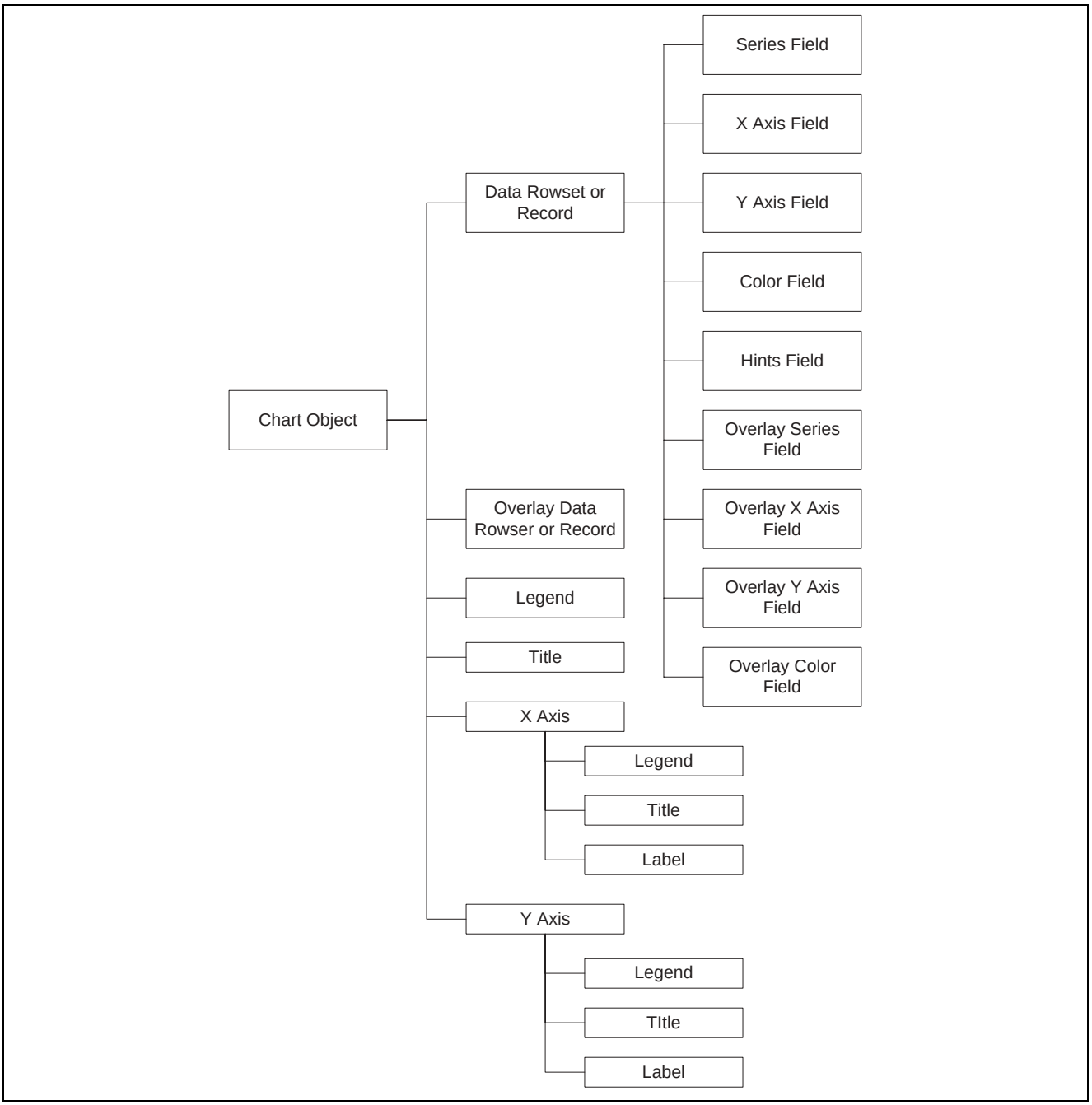
There is only one object used with the chart class, that is, a chart object. There are no subobjects. All methods and properties are used with the chart object. You do not have to instantiate a subobject to access its properties and methods.

However, for a conceptual overview, it's easier to view the chart object in pieces. Just remember that these pieces do *not*represent subobjects.

A chart contains the following major parts:

- X Axis
- Y Axis
- Overlay
- Legend
- Data
- Title

The X, Y, and Overlay axis have access to titles and labels associated with their data. The following is a conceptual overview of the chart class.



Overview of Chart class

Chart Terms

Charts can be composed of many different elements, such as a legend, a title, a label, and so on. The following is a list of general chart terms and their description:

Series A group of related information. For example, if you were tracking sales for several divisions over many years, each division could be a series.

	Generally, every series in a chart has a distinct color. You can have more than one series in a chart for comparison.
Overlay	Related data represented by a line drawn over the background chart. You can also specify a data series, with different line types for each overlay.
Legend	Text that identifies the different series in the chart.
Title	There are three titles: Main, X axis, and Y axis. These are text that identifies that portion of the chart.
Label	Text that identifies the data on one of the axis.
Y axis	The axis that contains the data. In most charts, this is the vertical axis. In a horizontal bar chart, this is the horizontal axis.
X axis	The axis that data is measured against.

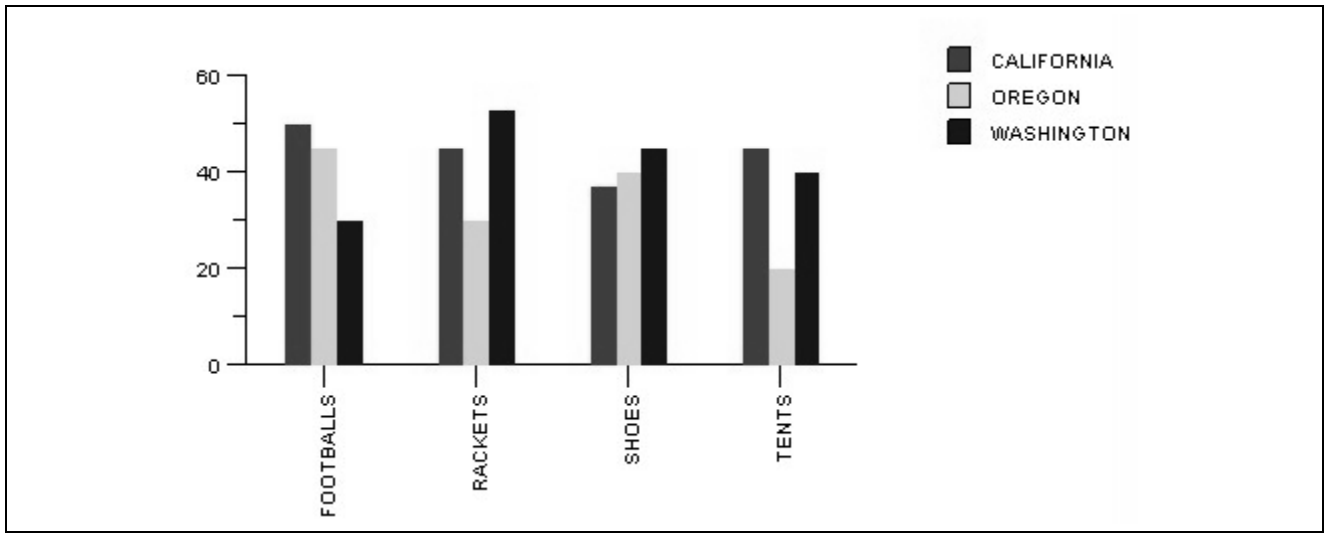
Chart Types

The following chart types are available for PeopleSoft Chart:

- 2D Bar.
- 2D Histogram.
- 2D Horizontal Bar.
- 2D Horizontal Percent Bar.
- 2D Horizontal Stacked Bar.
- 2D Line.
- 2D Percent Bar.
- 2D Pie.
- 2D Stacked Bar.
- 3D Bar.
- 3D Percent Bar.
- 3D Pie.
- 3D Stacked Bar.

2D Bar

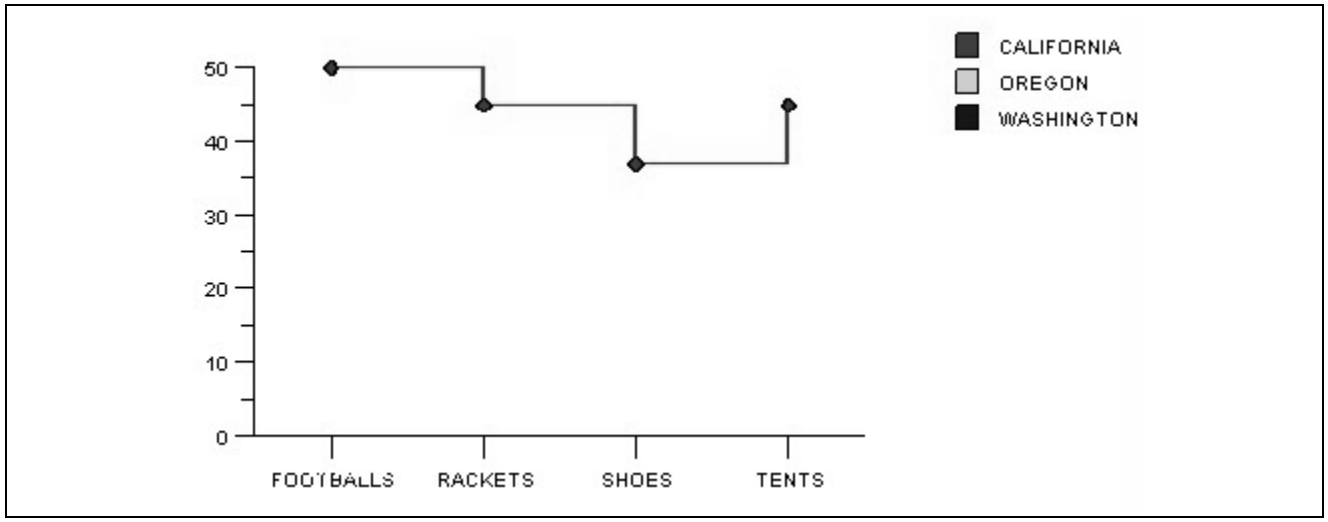
The following is an example of a two-dimensional bar chart.



Type=%ChartType_2DBar

2D Histogram

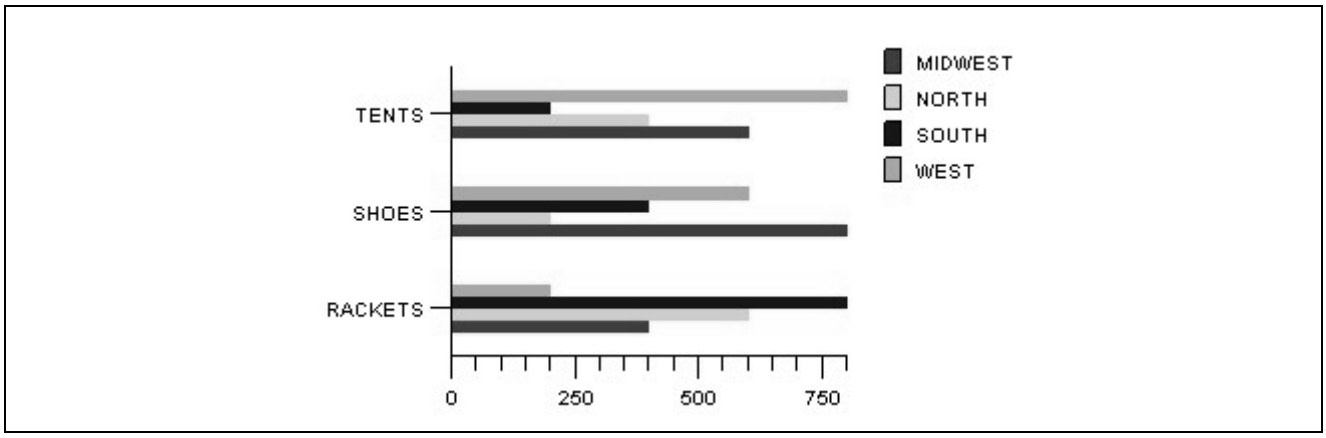
The following is an example of a two-dimensional histogram chart.



Type = %ChartType_2DHistogram

2D Horizontal Bar

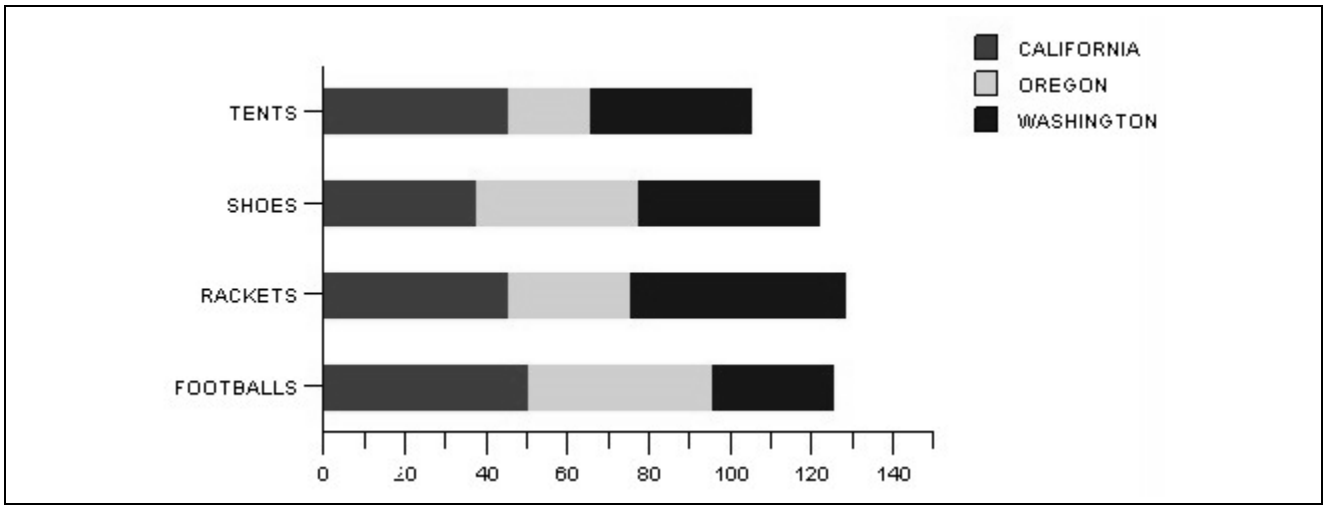
The following is an example of a two-dimensional horizontal bar chart.



Type = %ChartType_2DHorizBar

2D Horizontal Stacked Bar

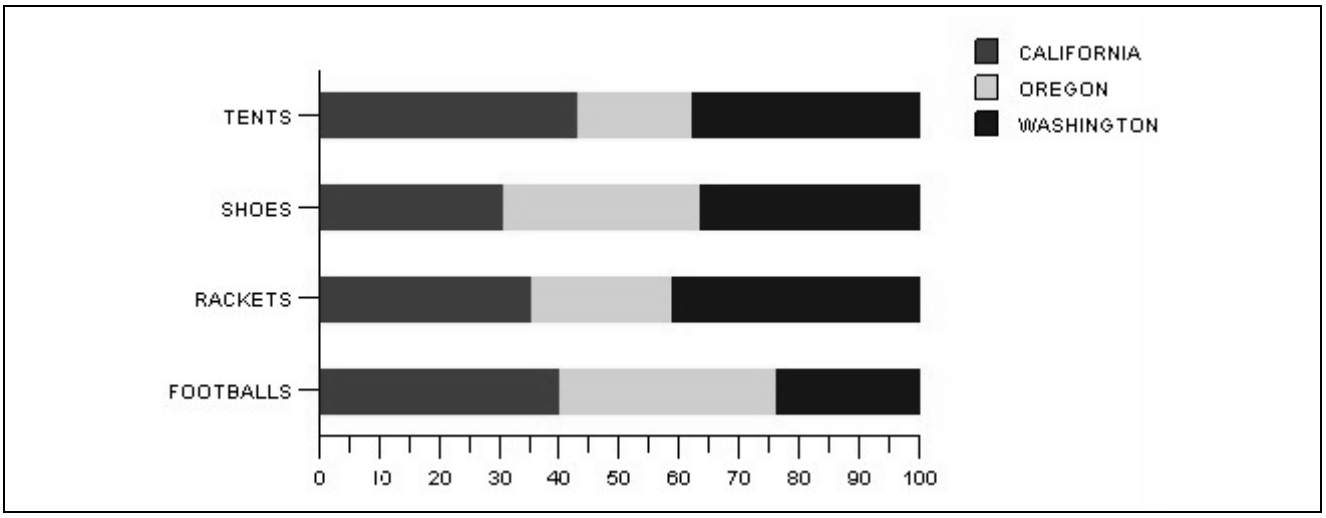
The following is an example of a two-dimensional horizontal stacked bar chart.



Type = %ChartType_2DHorizStackedBar

2D Horizontal Percent Bar

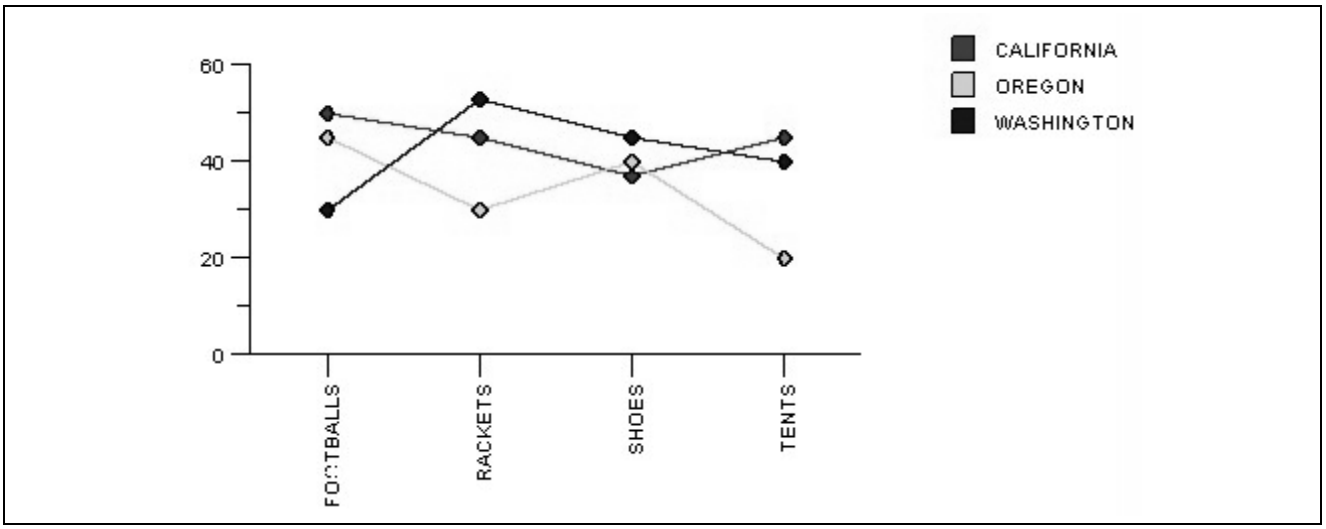
The following is an example of a two-dimensional horizontal percent bar chart.



Type = %ChartType_2DHorizPercentBar

2D Line

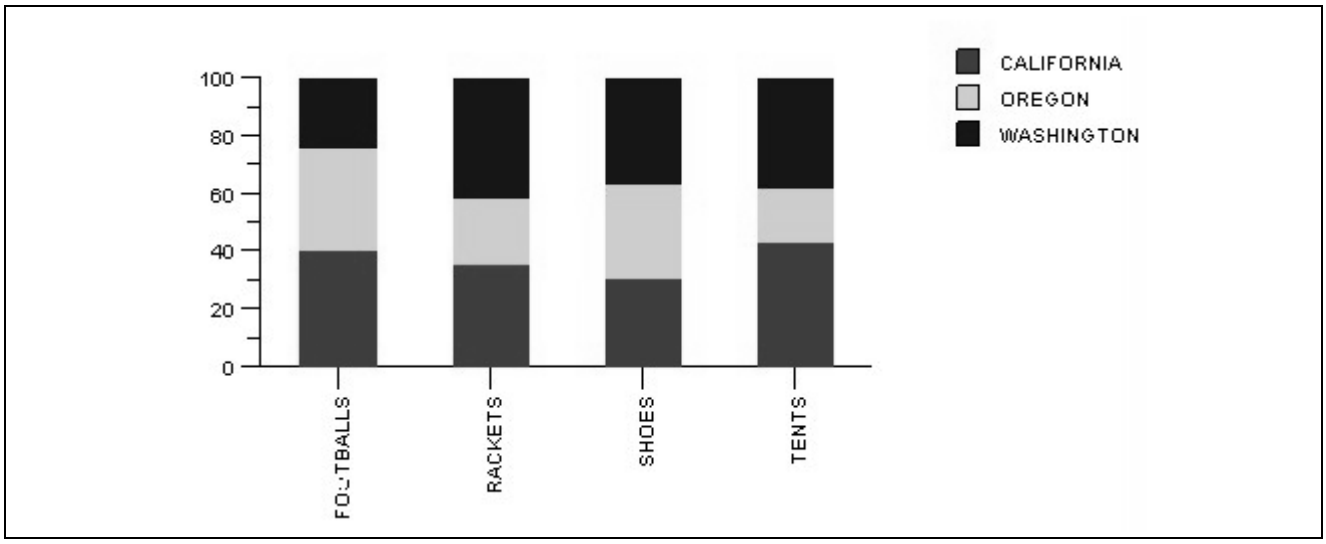
The following is an example of a two-dimensional line chart.



Type = %ChartType_2DLine

2D Percent Bar

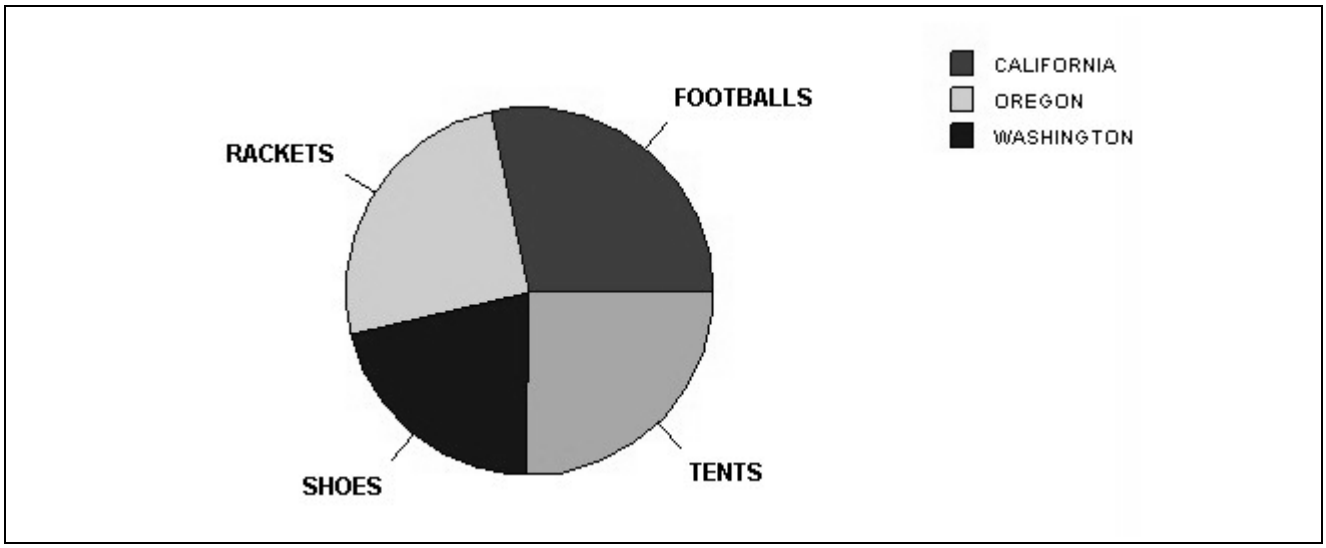
The following is an example of a two-dimensional percent bar chart.



Type = %ChartType_2DPercentBar

2D Pie

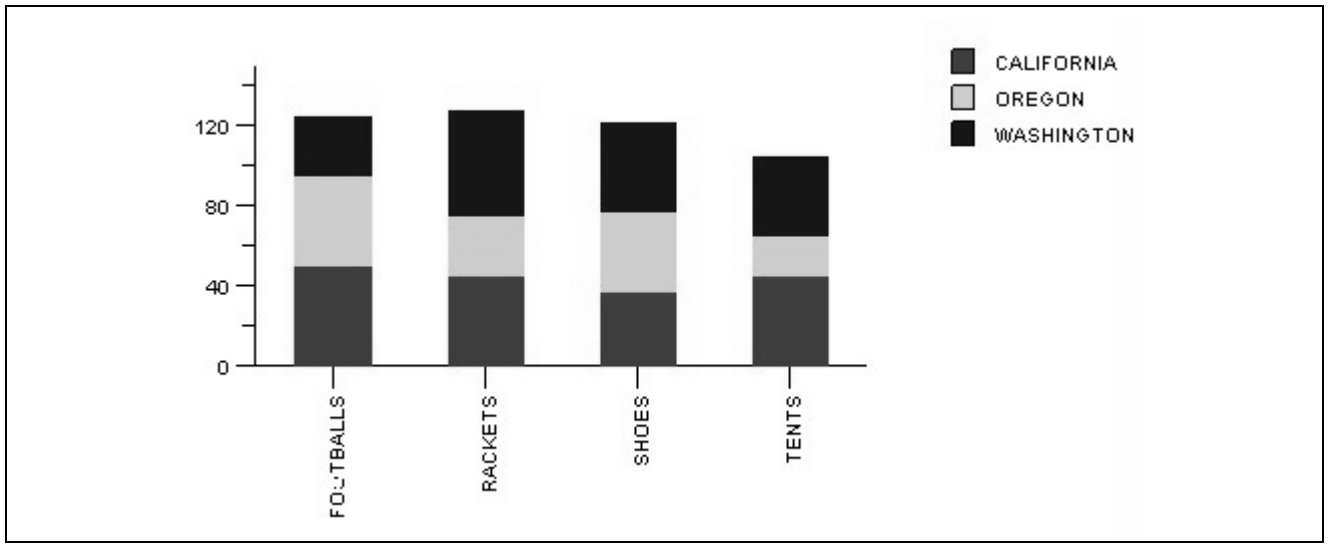
The following is an example of a two-dimensional pie chart.



Type = %ChartType_2DPie

2D Stacked Bar

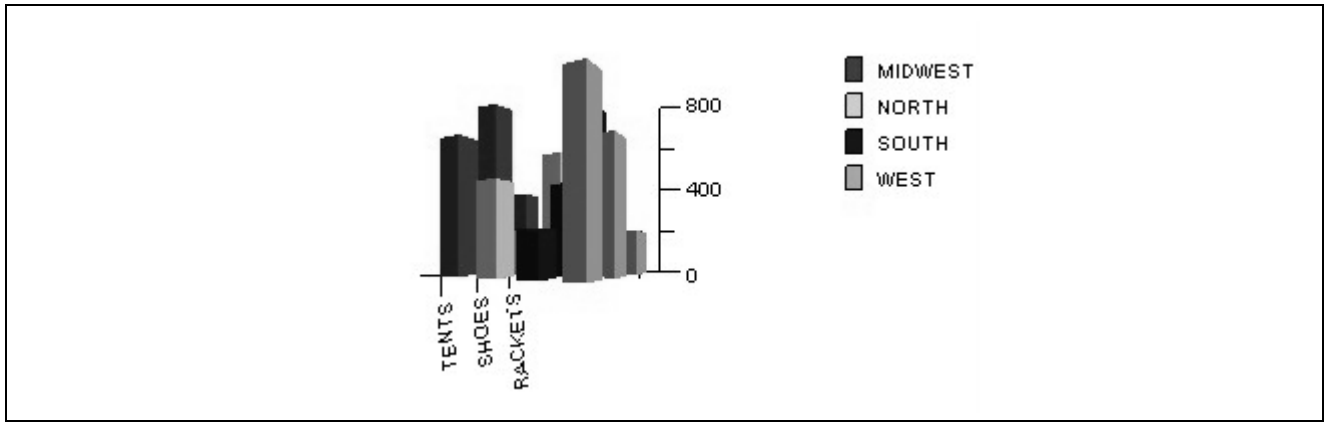
The following is an example of a two-dimensional stacked bar chart.



Type = %ChartType_2DStackedBar

3D Bar

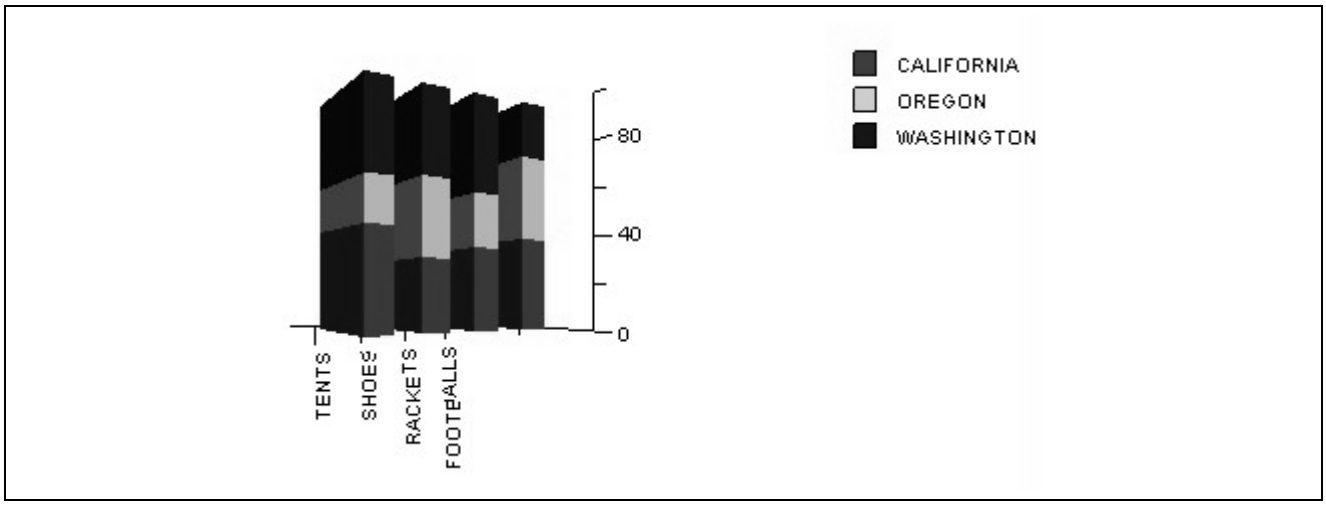
The following is an example of a three-dimensional bar chart.



Type = %ChartType_3DBar

3D Percent Bar

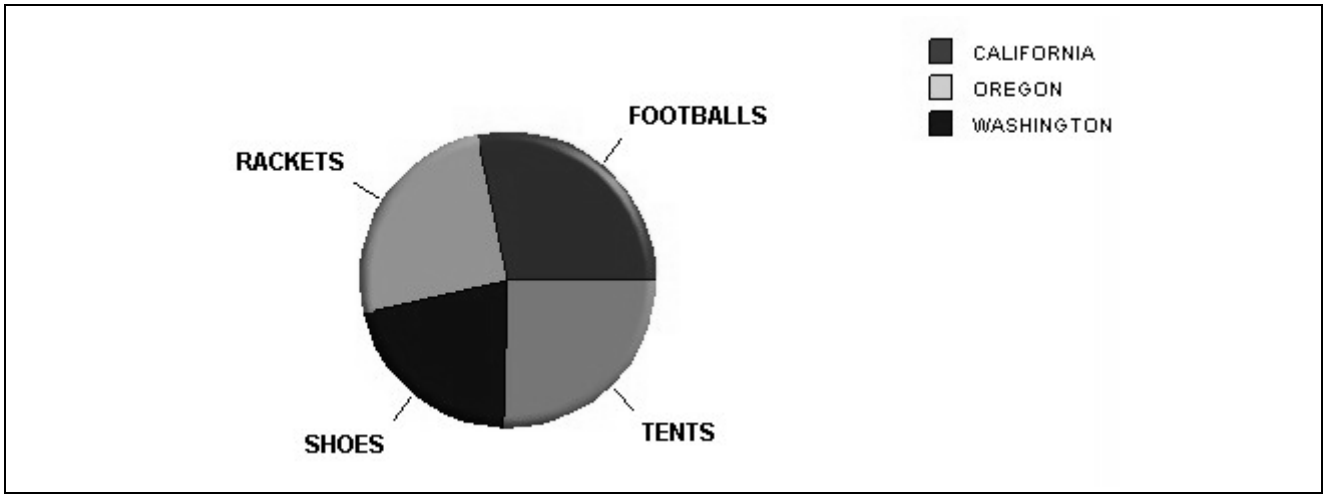
The following is an example of a three-dimensional percent bar chart.



Type = %ChartType_3DPercentBar

3D Pie

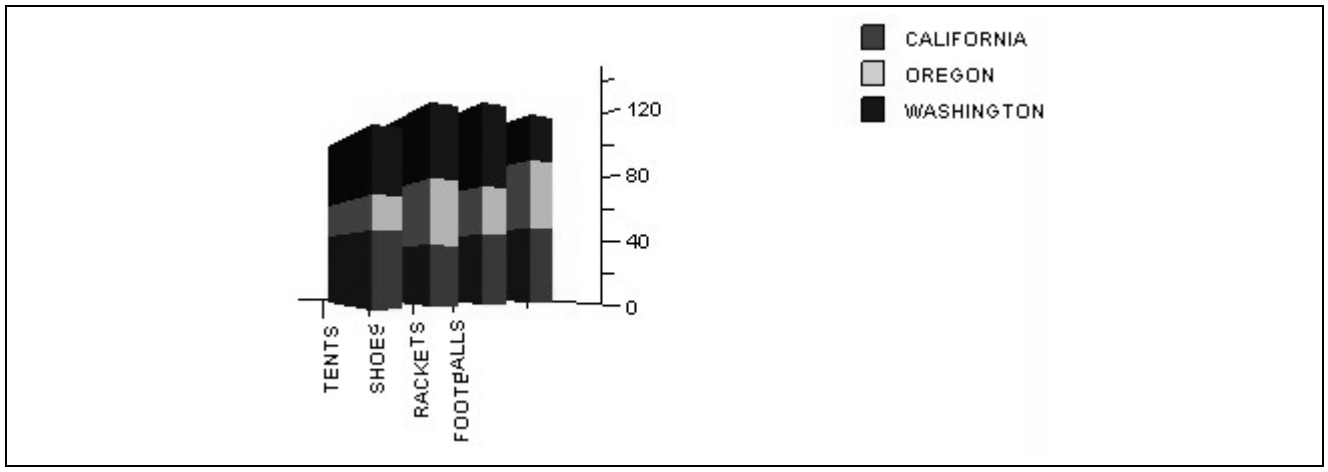
The following is an example of a three-dimensional pie chart.



Type = %ChartType_3DPie

3D Stacked Bar

The following is an example of a three-dimensional stacked bar chart.



Type = %ChartType_3DStackedBar

Style Sheets and Charts

A *style sheet* is a definition, like a record or field definition, that you create in PeopleSoft Application Designer. It contains a collection of formatting styles (called *classes*), each of which can be applied to either the entire chart or to specific elements within a chart.

You can use a style sheet, and the styles in a style sheet, for the different parts of a chart. You can either specify a style sheet for the entire chart (using either the chart **Style** or **StyleSheet** property) or for an element (using that element's **Style** property.) The override rules are:

1. If you specify the Style property for any element (XAxis, XAxisTitle, YAxis, YAxisTitle, Legend, MainTitle, and so on) the style is used.
2. If you don't specify the Style property for any element of a chart object, but you specify either the Style or StyleSheet property for the chart object itself, that style is used for the unspecified elements.
3. If you don't specify styles for any elements of a chart object, and you don't specify style for the chart object itself, a default style for any chart object is used.

Note. PeopleSoft recommends not using your own styles or style sheets unless absolutely necessary.

See Also

PeopleSoft Application Designer, Creating Style Sheet Definitions

Considerations Using Style Sheets

The following style sheet properties only are supported for PeopleSoft Chart:

- Font Family.
- Font Foreground Color.
- Font Size.
- Font Style (only Normal and Italic are supported.)

• Font Weight (only Normal and Bold are supported).

Default Style Sheet Classes

The default style sheet associated with a PeopleSoft Chart is PTSTYLEDEF. The default classes associated with this style sheet include:

Style Sheet Class Name	Associated Chart Element
PSCHARTDEFAULT	used with chart Style
PSCHARTTITLE	used with MainTitleStyle
PSCHARTDEFAULT	used with XAxisStyle
PSCHARTAXISTITLE	used with XAxisTitleStyle
PSCHARTDEFAULT	used with YAxisStyle
PSCHARTAXISTITLE	used with YAxisTitleStyle
PSCHARTDEFAULT	used with LegendStyle

Component Processor Considerations

Sometimes records at level 0 on a page are considered *work* records, even when they aren't specified as such in PeopleSoft Application Designer. Work records aren't updated by the database and are skipped when the component is run.

A level 0 record gets marked as a work record when:

• All the fields for the record are used as read-only fields.

• All the values for these fields can be read from the input keys.

When a chart field is attached to a level 0 record field of a record considered a work record, it's also skipped when the system determines that record's 'work' status.

See Also

PeopleCode Developer's Guide, Referencing Data in the Component Buffer

Translation Considerations

If you hardcode a value for a label, a title, a data hint, and so on, that value is *not* translated. Be sure to specify field values or message catalog entries that will be translated.

iScripts and Charts

PeopleSoft supports two types of charts:

- Charts generated using the chart control on a page.
- Charts generated at runtime and used in an iScript.

To build a chart at runtime, use the **CreateObject** function instead of the **GetChart** function. You can then use the **GetChartURL** Response class method to use the URL in your application.

The following is an example of building a chart using a standalone rowset:

```
Function IScript_GetChartURL()
local object &MYCHART;
local string &MYURL;

&MYCHART = CreateObject("Chart");
&MYCHART.SetData(xx);

/* xx will be a data row set */

&MYURL = %Response.GetChartURL(&MYCHART);

/* use &MYURL in your application */
...
End-Function;
```

See Also

Chapter 9, Chart Class , Creating a Chart Using an iScript , page 770

Error Handling

The PeopleCode program is terminated if a field or record is missing at runtime. If a valid record is specified but no data is found, the chart is replaced by the message " No chart data was set."

If the record is not found in the buffer, the message "Invalid chart data source" appears.

If the Java Runtime Environment (JRE) is missing, an error appears and is logged.

Data Type of a Chart Object

Chart objects are declared using the Chart data type. For example,

```
Local Chart &MyChart;

Component Chart &Abs_Hist_Chart;
```

Scope of a Chart Object

A chart object can be instantiated only from PeopleCode.

You can use this object only in PeopleCode programs that are associated with an online process, not in an Application Engine program, a message subscription, a Component Interface, and so on.

Chart Class Reference

The following sections contain the methods and properties used with a chart.

The reference is divided up among the different parts of a chart, however, all methods and properties are used with the chart object: there are no subobjects in a chart.

- Chart Data Methods and Properties: Used to set X and Y axes data.
- Chart Appearance Properties: Used to set the type of graph, the line style, and so on.
- Chart Color Methods: Used to set and manipulate colors for either a series or for each point.
- Chart Axis Methods and Properties: Used to further control the appearance of the axes.
- Chart Title Properties: Used to set the text and style of titles for the graph and the axes.
- Chart Legend Methods and Properties: Used for each series legend.
- Chart Overlay Methods and Properties: Used to set and manipulate overlay data.

Chart Class Built-in Functions

Chapter 1, PeopleCode Built-in Functions, CreateObject , page 101

Chapter 1, PeopleCode Built-in Functions, GetChart , page 239

Chapter 1, PeopleCode Built-in Functions, GetChartURL , page 240

Chart Data Methods

These methods are used to set X and Y axes data.

Refresh

Syntax

```
Refresh()
```

Description

Use the **Refresh** method if the underlying data of the chart has changed, and you want to update the chart.

Note. You don't need to use the **Refresh** method after setting a property or method of the chart itself, such as setting the starting point or changing colors. When a method or property is used, the refresh method is automatically called.

Parameters

None.

Returns

None.

Example

```
&MyChart.Refresh();
```

See Also

Chapter 9, Chart Class , DataStartRow, page 747, Chapter 9, Chart Class , DataWidth, page 747, Chapter 9, Chart Class , SetDataXAxis, page 745, Chapter 9, Chart Class , SetDataYAxis, page 746.

SetData

Syntax

```
SetData({Record_Name | &Rowset})
```

Description

Use the SetData method to specify where the data from the chart is coming from.

You can use either a record name or an already instantiated, populated rowset. In general, specify a record when building a chart from page data and a rowset when building a chart from a standalone rowset.

If you make a change to the underlying data of a chart, use the **Refresh** method to update the chart.

Parameters

Parameter	Description
Record_Name	Specify the name of a record to be used to populate the chart with data.
&Rowset	Specify the name of an already instantiated rowset to populate the chart with data. This is generally a standalone rowset.

Returns

None.

Example

The following is the minimum code you need to create a Peoplesoft chart.

```
Local Chart &MyChart;

&MyChart = GetChart(ABS_HIST.CHART);
&MyChart.SetData(ABS_HIST);
&MyChart.SetDataYAxis(ABS_HIST.Reason);
&MyChart.SetDataXAxis(ABS_HIST.Duration);
```

The following example creates a chart using a standalone rowset.

```
Function IScript_GetChartURL()
local object &MYCHART;
local string &MYURL;
local rowset &MYROWSET;

&MYCHART = CreateObject("Chart");

&MYROWSET = CreateRowset(Record.ABS_HIST);

&EmplID = %Emplid;

&MYROWSET.Fill("Where EMPLID :=1", &EmplID);

&MYCHART.SetData(&MYROWSET);
&MYCHART.SetDataXAxis(ABS_HIST.ABSENCE_TYPE);
&MYCHART.SetDataYAxis(ABS_HIST.DURATION_DAYS);

&MYURL = %Response.GetChartURL(&MYCHART);

/* use &MYURL in your application */
...
End-Function;
```

See Also

Chapter 9, Chart Class , SetDataXAxis, page 745, Chapter 9, Chart Class , SetDataYAxis, page 746.

SetDataSeries

Syntax

SetDataSeries(*Record_Name.Field_Name*)

Description

Use the **SetDataSeries** method to specify the name of the field containing the series values. Every distinct value in this field is considered a unique series.

The series is plotted in the order given by the record.

By default, the legend is populated by the value of this field.

If this value is Null, the system assumes there is only one series to plot.

Parameters

Parameter	Description
Record_Name.Field_Name	Specify the name of the record, and the field on that record, that contains the series values for the chart.

Returns

None.

Example

```
&MYCHART.SetDataSeries(MYRECORD.MYSERIES);
```

See Also

Chapter 9, Chart Class , SetDataXAxis, page 745, Chapter 9, Chart Class , SetDataYAxis, page 746.

SetDataHints

Syntax

SetDataHints(*Record_Name.Field_Name*)

Description

Use the **SetDataHints** method to specify the field that contains the text that displays as a data hint.

What displays for a data hint depends on the value specified for the data hints method, as well as the value specified for the series and the data method.

Series Data but No Data Hints

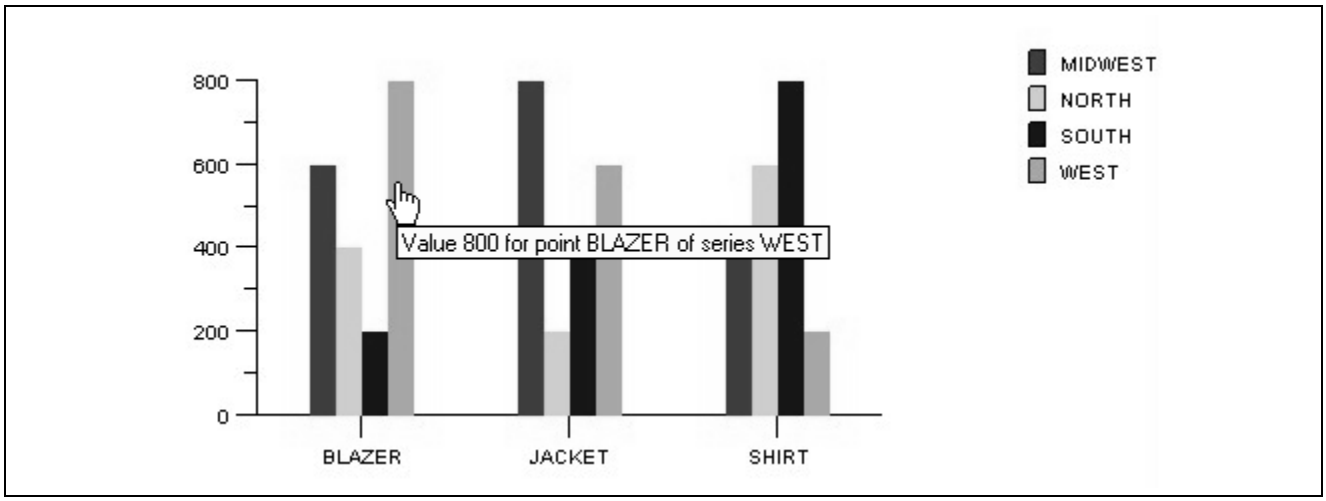
If you do *not* specify data hints, but you *do* specify both the data series and data methods, the hints appear as follows:

```
Value value for point x position of series series
```

Where:

- *value* is the Y data value
- *x position* is the X axis label
- *series* is the series value

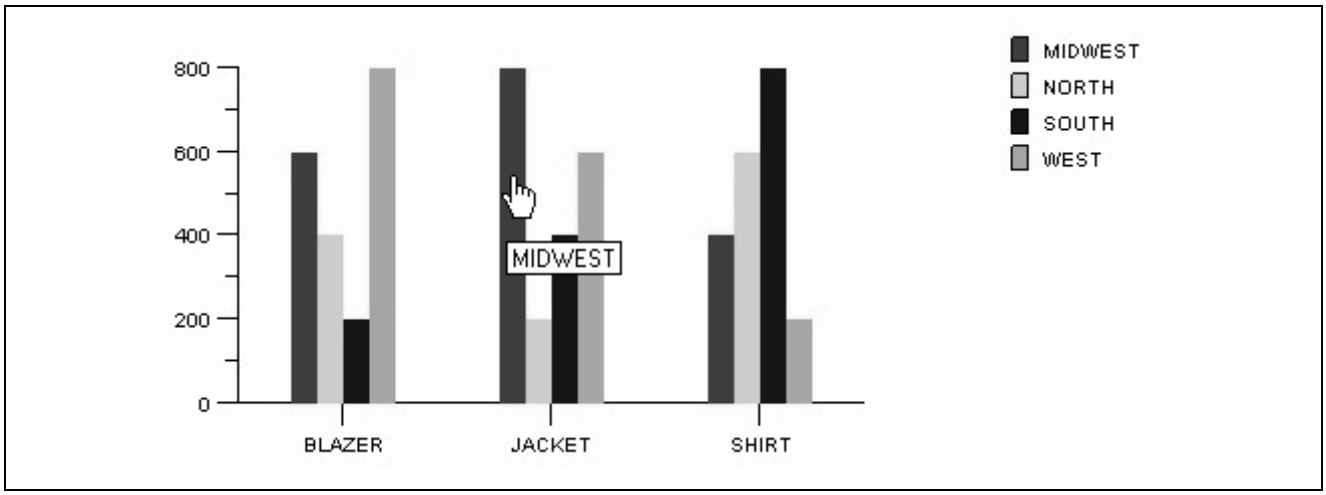
The following is an example of this type of data hints. The pop-up menu appears when the cursor hovers over any data value.



Data hints with multiple series example

Data Hints, Data, and Series

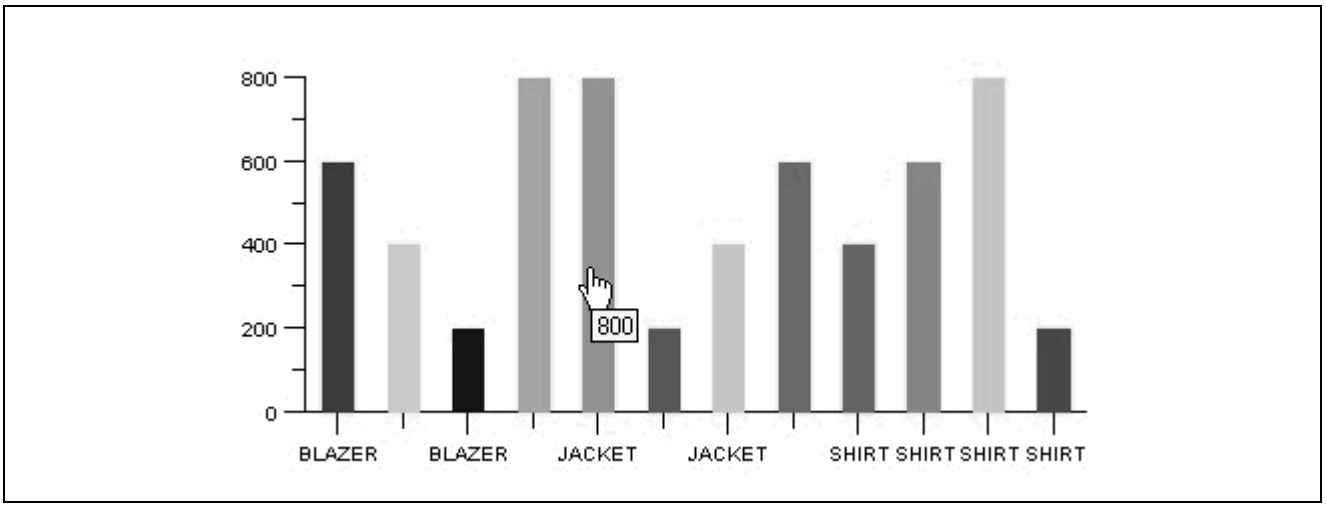
If you specify data hints, that is the value that appears. In the following example, the data hints are set to the same value as the series is set, that is, the region.



Data hints set to the region value

Data and Series, but no Data Hints

If you do *not* specify SetDataHints and do *not* specify SetDataSeries, the value of the Y axis appears in the data hint:



Data hints from default data

Parameters

Parameter	Description
Record_Name.Field_Name	Specify the name of the record, and the field on that record, that contains the data hints for the chart.

Returns

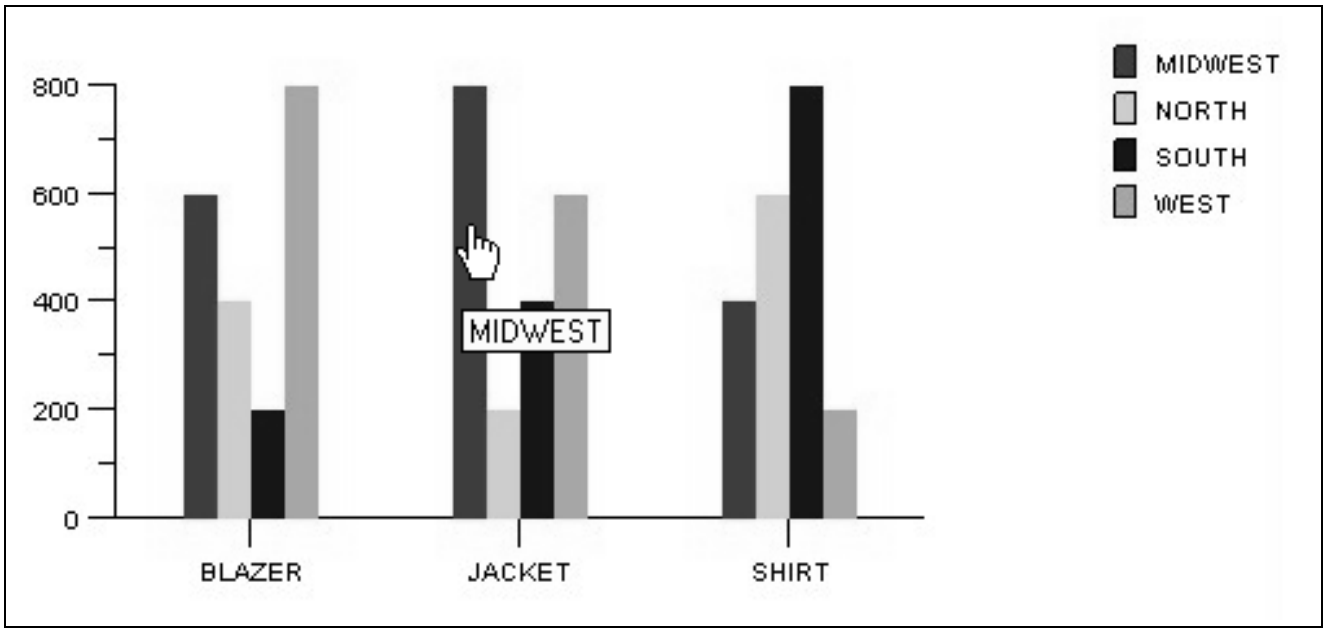
None.

Example

The following PeopleCode example:

```
&oChart.SetDataHints(QE_CHART_RECORD.QE_CHART_REGION);
```

Could produce data hints as follows:



Data hints set to the region value

See Also

Chapter 9, Chart Class , SetDataXAxis, page 745, Chapter 9, Chart Class , SetDataSeries, page 741.

SetDataURLs

Syntax

SetDataURLs(*FieldName*)

Description

Use the **SetDataURLs** method to specify a URL to be launched when a data point is clicked in the chart. The URL for each point must be given as a string, that is, enclosed with quotation marks. You can also specify *record.fieldname*. Use this method for charts created from a chart control or a standalone rowset.

One use of this method could be to launch a JavaScript that generates a pop-up menu.

Parameters

Parameter	Description
FieldName	Specify a field name that's used to populate the URL for each data point.

Returns

None.

Example

```
&MyChart.SetDataURLs(MyRecord.URL);
```

SetDataXAxis

Syntax

SetDataXAxis(*Record_Name.Field_Name*)

Description

Use the **SetDataXAxis** method to specify the groups along the X axis.

If this value is not set or Null, the Y values are plotted along the X axis in groups labeled by their order number.

The order of the data is plotted in the order of the data in the record.

By default, the labels along the X axis are populated by the value of this field.

Parameters

Parameter	Description
Record_Name.Field_Name	Specify the name of the record, and the field on that record, that contains the data for the X axis for the chart.

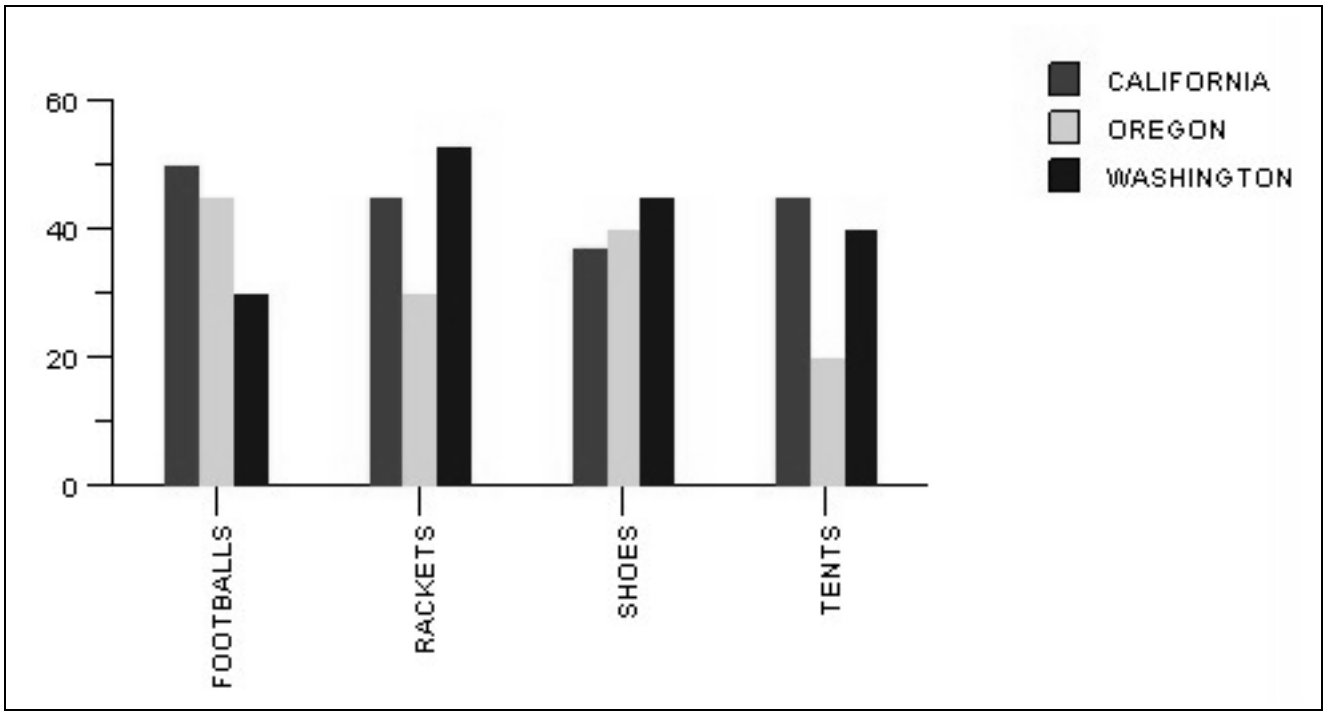
Returns

None.

Example

In the following example, the Y axis is set to a numeric amount, while the X axis is set to the products.

```
&oChart = GetChart(QE_CHART_DUMREC.QE_CHART_FIELD);

&oChart.SetData(Record.QE_CHART_RECORD);
&oChart.SetDataYAxis(QE_CHART_RECORD.QE_CHART_SALES);
&oChart.SetDataXAxis(QE_CHART_RECORD.QE_CHART_PRODUCT);
&oChart.SetDataSeries(QE_CHART_RECORD.QE_CHART_REGION);
```



X axis set with product data

See Also

Chapter 9, Chart Class , SetDataYAxis, page 746, Chapter 9, Chart Class , SetData, page 739.

SetDataYAxis

Syntax

SetDataYAxis(*Record_Name.Field_Name*)

Description

Use the **SetDataYAxis** to specify the data to plot in the chart.

The Y axis is always considered the data axis.

The Y axis must always be numeric. If the greatest Y axis value is less than ten, the Y axis labels are given a decimal place. If the greatest Y axis value is greater than or equal to ten, the Y axis labels are displayed as integers.

Note. If you've specified negative numbers in the Y axis, the X axis is drawn according to the **XAxisCross** property.

Parameters

Parameter	Description
Record_Name.Field_Name	Specify the name of the record, and the field on that record, that contains the data for the Y axis for the chart.

Returns

None.

Example

```
&oChart.SetDataYAxis(QE_CHART_RECORD.QE_CHART_SALES);
```

See Also

Chapter 9, Chart Class , SetDataXAxis, page 745, Chapter 9, Chart Class , SetData, page 739.

Chart Data Properties

These properties are used to set X and Y axes data.

DataStartRow

Description

This property sets or returns the row number (after any preprocessing) to start plotting at. This is useful when you have many rows of data in a rowset, and you want to start at a particular row. This can also be useful for creating your own 'scrolling' effect, by specifying both the start row and how many rows to be displayed (by using the DataWidth property.)

This property is read-write.

See Also

Chapter 9, Chart Class , DataWidth, page 747.

DataWidth

Description

This property returns or sets the number of rows to plot for a series. The starting point for the number of rows is set with the DataStartRow property.

This property is read-write.

Example

Suppose your rowset returned 400 rows. You wouldn't want all 400 to display in your chart. Instead, you'd pick a subset of those rows. The row to start with is set with DataStartRow, while how many rows to display is set with DataWidth.

The following code could be used with a pushbutton connected with a chart. The pushbutton enables the next set of data to be displayed with the chart. Note that you don't need to use the **Refresh** method with this code example. When a method or property is used, the refresh method is automatically called.

```
Local Chart &MyChart;

&MyChart = GetChart(ChartRec.DisplayField);
```

```
&Start = &MyChart.DataStartRow;
/* display the next 20 rows of data */
&MyChart.DataStartRow = &Start + 20;
&MyChart.DataWidth = 20;
```

See Also

Chapter 9, Chart Class , DataStartRow, page 747.

ImageMap

Description

This property returns the HTML client-side image map. The image map is what makes the chart interactive.

This property is read-only.

See Also

Chapter 9, Chart Class , Creating a Chart Using an iScript , page 770.

Chart Appearance Properties

These properties are used to set the type of graph, the line style, and so on.

GridLines

Description

Use this property to specify whether to show grid lines with the chart.

If you don’t specify a value, no grid lines are shown.

The values are:

Numeric Value	Constant Value	Description
0	%ChartGrid_None	Don’t show grid lines (default)
1	%ChartGrid_Horizontal	Show horizontal grid lines
2	%ChartGrid_Vertical	Show vertical grid lines
3	%ChartGrid_Both	Show both vertical and horizontal grid lines

This property is read-write.

Height

Description

Use this property to specify the height of the chart. This property takes a numeric value. The unit of measurement is pixels.

Generally this property is used with charts created for iScripts, to specify the height of the image, before generating the image map. You can also use it to overwrite the height set in PeopleSoft Application Designer.

If you try to read this property before setting it, the value returned is zero.

This property is read-write.

IsDrillable

Description

Use this property to specify whether the end-user can click on the chart and trigger a PeopleCode program in the FieldChange event for that row.

By default, when the chart data comes from a record, a PeopleCode program in the FieldChange event for the Y-Data field of the clicked row is executed.

If the data is in the form of a rowset, there is no FieldChange event. In this case, the URLs for *each data point* must be explicitly set by an extra field in the rowset and the property **SetDataURLs**.

This property takes a Boolean value: True if the end-user can click on the chart to initiate an action, False otherwise.

Note. Setting this property has no effect if the **IsPlainImage** property is set as True. Setting this property has no effect if the chart contains overlay data.

Note. When using charts in iScripts, this property must be used in conjunction with the **SetDataURLs** method. The SetDataURLs method must point to a field that is populated with the URLs to be triggered for each data point in the chart.

The default value for this property is False.

This property is read-write.

See Also

Chapter 9, Chart Class , IsPlainImage, page 749.

IsPlainImage

Description

Use this property to specify whether the chart is built without any extra HTML, such as SRC or MAP tags.

If this property is specified as True, the end-user won't be able to click on the chart to trigger an event. In addition, the pop-up data hints won't appear when the end-user passes a mouse over the chart. However, you may see a performance improvement if your chart has an extremely complicated image map and you specify this property as True.

This property takes a Boolean value: True if the chart is built without extra HTML, False otherwise.

The default value of this property is False.

This property is read-write.

LineType

Description

Use the **LineType** property to specify the style of series that are plotted as lines. This is for a regular series only. To specify the style of line for overlay data, use the OLLineType property.

This property takes either a numeric or constant value. The values are:

Numeric Value	Constant Value	Description
1	%ChartLine_Solid	Draw series with a solid line.
2	%ChartLine_Dash	Draw series with a line composed of dashes.
3	%ChartLine_Dot	Draw series with a line composed of dots.
4	%ChartLine_DashedDot	Draw series with a line composed of both dashes and dots.

This property is read-write.

See Also

Chapter 9, Chart Class , OLLineType, page 765.

RotationAngle

Description

Use this property to specify the rotation angle of the chart. This property is valid only with 3D style charts, that is, charts that have Type set as %ChartType_3DBar, %ChartType_3DStackedBar, and so on.

Note. This property has no effect on the 3D pie chart.

If you change the value of this property, you don't have to use the Refresh method to refresh the chart.

This property takes a numeric value from 1 to 360.

This property does not take negative numbers.

The chart is rotated in a clockwise direction.

This property is read-write.

Style

Description

A *style sheet* is a definition, like a record or field definition, that you create in PeopleSoft Application Designer. It contains a collection of formatting styles, called *classes*, each of which can be applied to either the entire chart or to specific elements within a chart.

Use this property to specify the overall style of the chart by specifying a style sheet class. The value must be a valid class within the style sheet specified for the chart.

Individual style properties (such as SetXAxisStyle, MainTitleStyle, and so on) can be used to override this value for individual elements of the chart.

This property is read-write.

StyleSheet

Description

Use this property to associate a style sheet with a chart.

The PTSTYLEDEF style sheet is the default style sheet associated with a chart.

Both the Style property for the chart, and individual style properties (such as SetXAxisStyle, MainTitleStyle, and so on) can be used to override this value for individual elements of the chart.

This property is read-write.

Type

Description

Use this property to specify the type of chart that you want. You can specify either a numeric or constant value for this property. The valid values are:

Numeric Value	Constant Value	Description
0	%ChartType_2DBar	Two-dimensional bar chart
1	%ChartType_2DStackedBar	Two-dimensional stacked bar chart
2	%ChartType_2DPercentBar	Two-dimensional percent bar chart
3	%ChartType_2DHorizStackedBar	Two-dimensional stacked horizontal bar chart
4	%ChartType_2DLine	Two-dimensional line chart
5	%ChartType_2DHistogram	Two-dimensional histogram chart
6	%ChartType_2DPie	Two-dimensional pie chart
7	%ChartType_3DBar	Three-dimensional bar chart
8	%ChartType_3DStackedBar	Three-dimensional stacked bar chart
9	%ChartType_3DPie	Three-dimensional pie chart
10	%ChartType_2DHorizPercentBar	Two-dimensional horizontal percent chart
11	%ChartType_3DPercentBar	Three-dimensional percent bar char
12	%ChartType_2DHorizBar	Two-dimensional horizontal bar chart

This property is read-write.

See Also

Chapter 9, Chart Class , Chart Types , page 728.

Width

Description

Use this property to specify the width of the chart. This property takes a numeric value. The unit of measurement is pixels.

Generally this property is used with charts created for iScripts, to specify the width of the image, before generating the image map. You can also use it to overwrite the height set in PeopleSoft Application Designer.

If you try to read this property before setting it, the value returned is zero.

This property is read-write.

Chart Colors

The following lists all the values you can use with the chart color methods. You can use either the numeric or constant value.

Numeric Value	Constant Value	Description
-1	%ChartColor_Series_Default	Default series color. When a data point is set to this, the default series color or the color that has been set by the user for that series is used instead. This value works only with the **SetColorArray**method.
0	%ChartColor_Black	Black
1	%ChartColor_Blue	Blue
2	%ChartColor_Cyan	Cyan
3	%ChartColor_DarkGray	DarkGray
4	%ChartColor_Gray	Gray
5	%ChartColor_Green	Green
6	%ChartColor_LightGray	LightGray
7	%ChartColor_Magenta	Magenta
8	%ChartColor_Orange	Orange

Numeric Value	Constant Value	Description
9	%ChartColor_Pink	Pink
10	%ChartColor_Red	Red
11	%ChartColor_White	White
12	%ChartColor_Yellow	Yellow
13	%ChartColor_Red_Orange	Red_Orange
14	%ChartColor_Yellow_Green	Yellow_Green
15	%ChartColor_Blue_Violet	Blue_Violet
16	%ChartColor_Purple	Purple
17	%ChartColor_Yellow_Orange	Yellow_Orange

Chart Color Methods

These methods are used to set and manipulate colors for either a series or for each point.

SetColorArray

Syntax

SetColorArray(*&Array_of_Color*)

Description

Use the **SetColorArray** method to set the colors for a series. You can specify only a one-dimensional array for the parameter. This method applies only to base chart data, not to the overlay data.

The type of array can be any of the following:

- String
- Number
- Integer
- Decimal

SetDataColor does not work with line charts. Individual line segments can not be colored as there are less line segments than points. **SetColorArray** must be used instead to color each complete line (series).

Parameters

Parameter	Description
&Array_of_Color	Specify an already instantiated array that contains the color values that you want for the series. You can specify either a constant or a number.

See Chapter 9, Chart Class , Chart Colors, page 752.

Returns

None.

Example

The following example sets the four series in the chart:

```
&NumArray = CreateArray(1, 3, 6, 9);

&oChart.SetColorArray(&NumArray);
```

See Also

Chapter 7, Array Class , page 633.

SetDataColor

Syntax

SetDataColor(*Record_Name.Field_Name*)

Description

Use the **SetDataColor** method to specify a field that contains the color for each data point. Each field for each datapoint must contain either a number or a constant, specifying the color you want to use.

SetDataColor does not work with line charts. Individual line segments can not be colored as there are less line segments than points. **SetColorArray** must be used instead to color each complete line (series).

Note. This method has no effect when used with an overlay.

Parameters

Parameter	Description
Record_Name.Field_Name	Specify the name of the record, and the field on that record, that contains the colors for each data point. You can specify either a constant or a number.

See Chapter 9, Chart Class , Chart Colors, page 752.

Returns

None.

Chart Axes Methods

These methods are used to further control the appearance of the axes.

SetXAxisLabels

Syntax

SetXAxisLabels(*&Array_of_Any*)

Description

Use the **SetXAxisLabels** method to specify an array of labels for the X axis. By default, the labels are populated from the record field specified by **SetDataXAxis**. Labels specified with SetXAxisLabels overwrite the default labels.

The X axis always appears at the bottom of the chart, even with negative values.

Note. Default label text is automatically translated. If you set your own labels, be sure to use translated text, such as message catalog entries.

Parameters

Parameter	Description
&Array_of_Any	Specify an already instantiated array of any that contain the labels that you want to use for the X Axis.

See Also

Chapter 9, Chart Class , SetDataXAxis, page 745, Chapter 7, Array Class , page 633.

SetYAxisLabels

Syntax

SetYAxisLabels(*&Array_of_Any*)

Description

Use the **SetYAxisLabels** method to specify an array of labels for the Y axis. Labels specified with SetYAxisLabels overwrite the default labels.

If the greatest Y axis value is less than ten, the default Y axis labels are given a decimal place. If the greatest Y axis value is greater than or equal to ten, the default Y axis labels are displayed as integers.

The X axis always appears at the bottom of the chart, even with negative values.

Note. Default label text is automatically translated. If you set your own labels, be sure to use translated text, such as message catalog entries.

If you set the labels yourself, you may need to set the YAxisTicks and YAxisMax because you no longer have numbers that correlate to the data points.

Parameters

Parameter	Description
&Array_of_Any	Specify an already instantiated array of any that contain the labels that you want to use for the Y Axis.

See Also

Chapter 9, Chart Class , SetDataXAxis, page 745, Chapter 9, Chart Class , YAxisTicks, page 759, Chapter 9, Chart Class , YAxisMax, page 758, Chapter 7, Array Class , page 633.

Chart Axes Properties

These properties are used to further control the appearance of the axes.

XAxisCross

Description

Use this property to specify the appearance of the X axis data when there are both negative and positive values.

You can specify either a numeric value or a constant for this property. Values are:

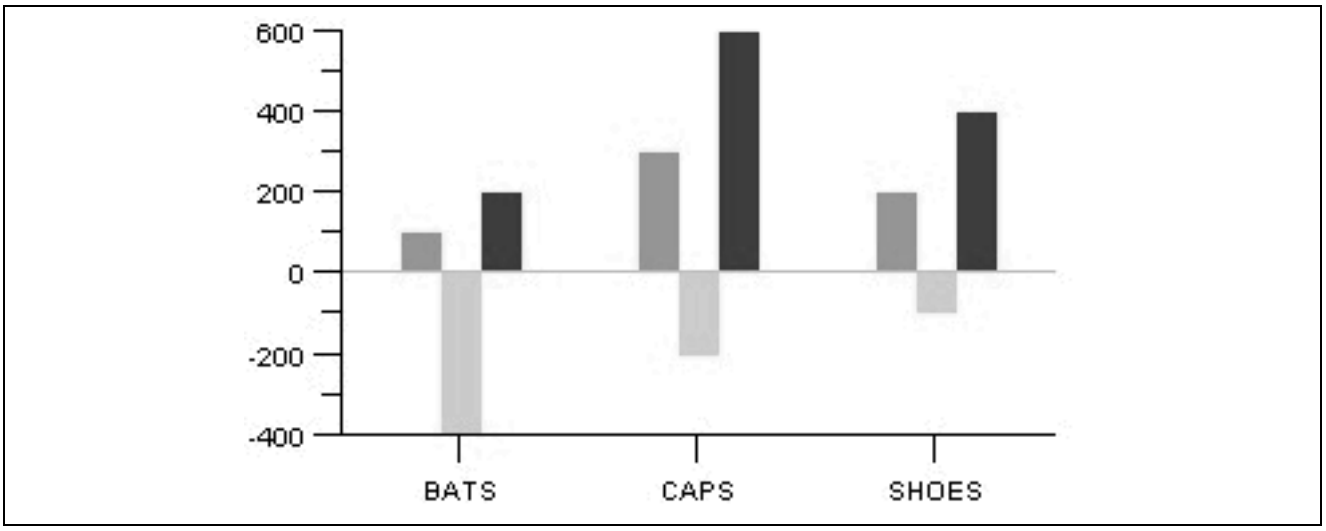
Numeric Value	Constant Value	Description
0	%ChartXAxisCross_Zero	The X axis extrudes from the zero point on the Y axis.
–1	%ChartXAxisCross_Bottom	The X axis extrudes from the Y minimum on the Y axis.
–2	%ChartXAxisCross_ZeroLineBttm	The X axis extrudes from the Y minimum on the Y axis, and a line is drawn from the zero point on the Y axis.

Example

The following code:

```
&MyChart.XAxisCross = %ChartXAxisCross_ZeroLineBttm;
```

produces a chart similar to this:



Example of %ChartXAxisCross_ZeroLineBttm

XAxisLabelOrient

Description

Use this property to specify whether the X axis label is displayed horizontally or vertically.

The default value is horizontal.

You can specify either a numeric value or a constant for this property. Values are:

Numeric Value	Constant Value	Description
1	%ChartText_Horizontal	The X axis label is displayed horizontally.
2	%ChartText_Vertical	The X axis label is displayed veritically This property is read-write.

XAxisStyle

Description

Use this property to specify the style of the X axis. The values for this property are the style classes contained in the style sheet associated with the chart.

Note. Pie chart labels take their style from this property.

This property is read-write.

See Also

Chapter 9, Chart Class , Style Sheets and Charts , page 735.

YAxisLabelOrient

Description

Use this property to specify whether the Y axis label is displayed horizontally or vertically.

The default value is horizontal.

You can specify either a numeric value or a constant for this property. Valid values are:

Numeric Value	Constant Value	Description
1	%ChartText_Horizontal	The Y axis label is displayed horizontally.
2	%ChartText_Vertical	The Y axis label is displayed veritically

This property is read-write.

YAxisMax

Description

Use this property to specify the maximum value of the Y (data) axis. This property takes a numeric value.

The maximum value sets the largest value that is displayed on the axis.

This property is read-write.

YAxisMin

Description

Use this property to specify the minimum value of the X axis. This property takes a numeric value.

The minimum value sets the point at which the chart plots in positive and negative directions.

The default minimum value is the default value of the Y axis.

This property is read-write.

YAxisStyle

Description

Use this property to specify the style of the Y axis. The values for this property are the style classes contained in the style sheet associated with the chart.

This property is read-write.

See Also

Chapter 9, Chart Class , Style Sheets and Charts , page 735.

YAxisTicks

Description

Use this property to specify the number of units along the Y (data) axis. This property takes a numeric value.

This property is read-write.

Chart Title Properties

These properties are used to set the text and style of titles for the graph and the axes.

MainTitle

Description

Use this property to specify the text for the main title of the chart.

This property takes a string value.

This property is read-write.

MainTitleOrient

Description

Use this property to specify the orientation of the text for the main title.

The default value is horizontal.

You can specify either a number or a constant for this property. The values are:

Numeric Value	Constant Value	Description
1	%ChartText_Horizontal	Title text is displayed horizontally.
2	%ChartText_Vertical	Title text is displayed vertically. This property is read-write.

MainTitleStyle

Description

Use this property to specify the style of the main title. The values for this property are the style classes contained in the style sheet associated with the chart.

This property is read-write.

See Also

Chapter 9, Chart Class , Style Sheets and Charts , page 735.

XAxisTitle

Description

Use this property to specify the text for the X axis title.

This property is read-write.

XAxisTitleOrient

Description

Use this property to specify the orientation of the text for the X axis title.

You can specify either a number or a constant for this property. The values are:

Numeric Value	Constant Value	Description
1	%ChartText_Horizontal	Title text is displayed horizontally.
2	%ChartText_Vertical	Title text is displayed vertically. This property is read-write.

XAxisTitleStyle

Description

Use this property to specify the style of the X axis title. The values for this property are the style classes contained in the style sheet associated with the chart.

This property is read-write.

See Also

Chapter 9, Chart Class , Style Sheets and Charts , page 735.

YAxisTitle

Description

Use this property to specify text of the title for the Y axis.

This property is read-write.

YAxisTitleOrient

Description

Use this property to specify the orientation of the text for the Y axis title.

You can specify either a number or a constant for this property. The values are:

Numeric Value	Constant Value	Description
1	%ChartText_Horizontal	Title text is displayed horizontally.
2	%ChartText_Vertical	Title text is displayed vertically. This property is read-write.

YAxisTitleStyle

Description

Use this property to specify the style of the Y axis title. The values for this property are the style classes contained in the style sheet associated with the chart.

This property is read-write.

See Also

Chapter 9, Chart Class , Style Sheets and Charts , page 735.

Chart Legend Methods

These methods are used for each series legend.

SetLegend

Syntax

SetLegend(*&Array_of_String*)

Description

Use the **SetLegend** method to specify alternative legends. By default, the legends are populated from the record field specified with **SetDataSeries**.

This method is used to write both the overlay legend and the series legend. The labels are overwritten in the order of elements in the array, that is, the first element overwrites the first series, the second overwrites the second, and so on, with the last element in the array overwriting the overlay legend. There can be more than one series in an overlay.

If you don't specify an element for an array (that is, a blank or a null) then no legend is listed for that series.

Note. Default label text is automatically translated. If you set your own labels, be sure to use translated text, such as message catalog entries.

Parameters

Parameter	Description
&Array_of_String	Specify an already instantiated array of string, containing the text that you want to use for the legend.

Returns

None.

Example

The following example displays a legend containing only the overlay.

```
&LegendArray = CreateArray("", "", "Revenue");

&MyChart.SetLegend(&LegendArray);
```

See Also

Chapter 9, Chart Class , SetDataSeries, page 741, Chapter 7, Array Class , page 633.

Chart Legend Properties

These properties are used for each series legend.

HasLegend

Description

Use this property to specify if a legend is displayed with the chart. This property takes a Boolean value: True, if the legend is displayed with the chart, False otherwise.

The default value is False.

This property is read-write.

LegendMaxEntries

Description

Use this property to specify the number of legend entries before a new column (or row) is created.

When the legend position is specified as left or right, the legend is drawn vertically so a new column is created when this property is exceeded.

When the legend position is specified as top or bottom, the legend is drawn horizontally and a new row is created when this property is exceeded.

The default value of this property is five.

This property is read-write.

LegendPosition

Description

Use this property to specify where the legend should appear, in relationship to the chart. You can specify either a numeric or constant value for this property.

Note. ChartLegend_Separate generates a legend without a chart. The legend takes over the entire charting area. The legend appears in the center of the chart.

The values are:

Numeric Value	Constant Value	Description
1	%ChartLegend_Left	Display legend to the left of the chart.
2	%ChartLegend_Right	Display legend to the right of the chart.
3	%ChartLegend_Top	Display legend on top of the chart.
4	%ChartLegend_Bottom	Display legend below the chart.
5	%ChartLegend_Separate	Generate a legend without a chart. This property is read-write.

LegendStyle

Description

Use this property to specify the style of the legend. The values for this property are the style classes contained in the style sheet associated with the chart.

This property is read-write.

See Also

Chapter 9, Chart Class , Style Sheets and Charts , page 735.

Chart Overlay Methods

These methods are used to set and manipulate overlay data.

SetOLData

Syntax

```
SetOLData({Record_Name | &Rowset})
```

Description

Use the SetOLData method to specify where the data for the overlay is coming from. If you make a change to the underlying data of a chart, use the **Refresh** method to update the chart.

There are no default values for the overlay data. You must specify the data.

It is possible to specify two different Y axis fields, however, you must make certain that different data plotted against the same axis makes sense. For example, plotting two dollar amounts would probably be okay. However, plotting dollars against the number of sales, and sharing the same axis, though technically it would work, it wouldn't look correct.

Parameters

Parameter	Description
Record_Name	Specify the name of a record to be used to populate the overlay of the chart with data.
&Rowset	Specify the name of an already instantiated rowset to populate the overlay of the chart with data.

See Also

Chapter 9, Chart Class , SetDataXAxis, page 745, Chapter 9, Chart Class , SetDataYAxis, page 746, Chapter 9, Chart Class , SetData, page 739.

SetOLDataSeries

Syntax

SetOLDataSeries(*Record_Name.Field_Name*)

Description

Use the **SetOLDataSeries** method to specify the name of the field containing the overlay series values. Every distinct value in this field is considered a unique series.

The series is plotted in the order given by the rows in the record.

If this value is Null, the system assumes there is only one series to plot.

Parameters

Parameter	Description
Record_Name.Field_Name	Specify the name of the record, and the field on that record, that contains the series values of the overlay for the chart.

See Also

Chapter 9, Chart Class , SetDataSeries, page 741.

SetOLDataXAxis

Syntax

SetOLDataXAxis(*Record_Name.Field_Name*)

Description

Use the **SetOLDataXAxis** method to specify the groups along the X axis if the axis is non-numeric for the overlay.

If the X axis is numeric, this method is used to give the position of the points along the axis. Only discrete values are supported for the X axis.

If this value is Null, the Y values are plotted along the X axis in groups labeled by their order number.

The order of the data is plotted in the order of rows in the record.

Parameters

Parameter	Description
Record_Name.Field_Name	Specify the name of the record, and the field on that record, that contains the data for the X axis for the overlay of the chart.

SetOLDataYAxis

Syntax

SetOLDataYAxis(*Record_Name.Field_Name*)

Description

Use the **SetOLDataYAxis** to specify the data to plot for the overlay of the chart.

This value must always be numeric.

There is no default value for this method.

Parameters

Parameter	Description
Record_Name.Field_Name	Specify the name of the record, and the field on that record, that contains the data for the overlay of the chart.

See Also

Chapter 9, Chart Class , SetDataXAxis, page 745, Chapter 9, Chart Class , SetDataYAxis, page 746.

Chart Overlay Properties

These properties are used to set and manipulate overlay data.

OLLineType

Description

Use the **OLLineType** property to specify the style of an overlay line.

This property takes either a numeric or constant value. The values are:

Numeric Value	Constant Value	Description
1	%ChartLine_Solid	Draw series with a solid line.
2	%ChartLine_Dash	Draw series with a line composed of dashes.
3	%ChartLine_Dot	Draw series with a line composed of dots.
4	%ChartLine_DashedDot	Draw series with a line composed of both dashes and dots. This property is read-write.

See Also

Chapter 9, Chart Class , LineType, page 750.

OLType

Description

Use this property to specify the style of the overlay line.

You can specify either a numeric or a constant value for this property. The values are:

Numeric Value	Constant Value	Description
4	%ChartType_2DLine	Overlay is drawn as a 2D line.
5	%ChartType_2DHistogram	Overlay is drawn as a 2D histogram.

This property is read-write.

Chart Examples

The following are some example charts, with the PeopleCode used to create them.

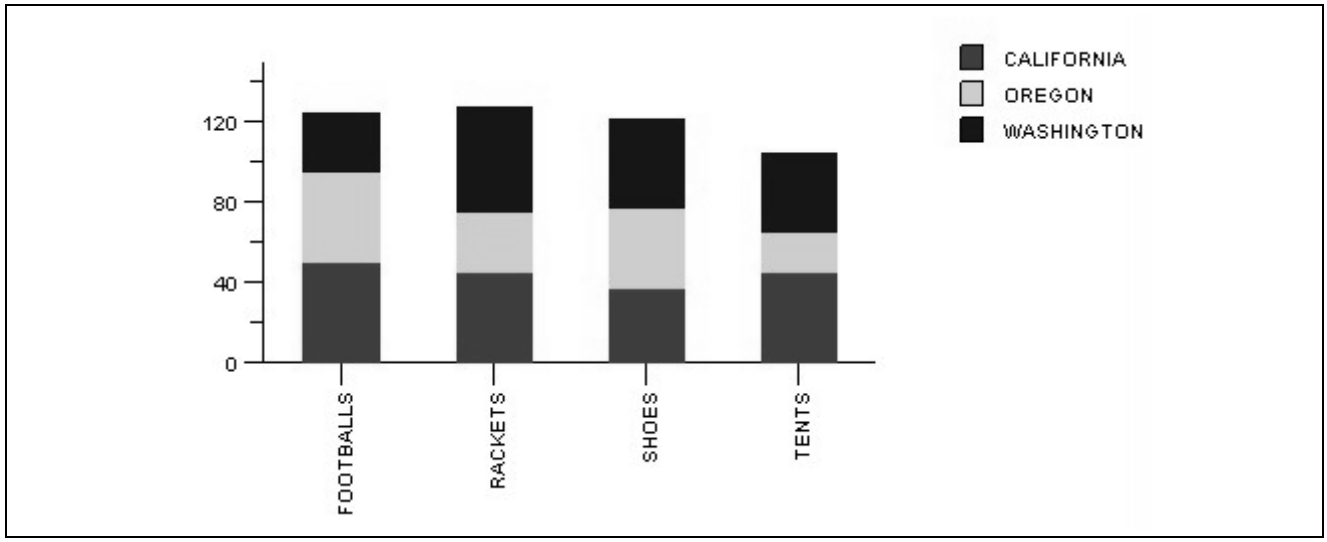
Creating a Chart

In the following example, the chart is created from a record with three fields, relating to sales, month, and product. The series in the chart are based on the product. The following is the complete code sample: the steps explain each line.

```
Global Chart &oChart;

&oChart = GetChart(QE_CHART_DUMREC.QE_CHART_FIELD);

&oChart.SetData(Record.QE_CHART_RECORD);
&oChart.SetDataYAxis(QE_CHART_RECORD.QE_CHART_SALES);
&oChart.SetDataXAxis(QE_CHART_RECORD.QE_CHART_PRODUCT);

&oChart.SetDataSeries(QE_CHART_RECORD.QE_CHART_REGION);
```

```
&oChart.Type = %ChartType_2DStackedBar;

&oChart.HasLegend = True;
&oChart.LegendPosition = %ChartLegend_Right;

&oChart.XAxisLabelOrient = %ChartText_Vertical;
```

The following is the chart produced by the previous code:



Test Chart - stacked bar

To create a chart:

1. Open PeopleSoft Application Designer.
2. Open the page where the chart is to be inserted.
3. Insert the chart control by either:
   - Clicking on the chart icon in the toolbar, or
   - Selecting Insert, Chart
4. Draw the chart control on the page.
5. Associate the chart control with a record field

   Every chart control must be associated with a record field. This is just the field for the chart control. It is *not* the field used for drilling down for data in the chart.

   Bring up the chart control properties by either
   - Double-clicking on the chart control, or
   - Right-clicking on the chart control and selecting Page Field Properties.
6. Select the record name and field for the chart.

   On the Record tab of the chart control properties, select the record and field for the chart.

   To make the control a page anchor, select the Enable as Page Anchor on the General tab.
7. Write your PeopleCode.

In some event on the page, such as Activate, put the PeopleCode you need for populating the chart.

8. Get the chart.

   The first thing you must do is get the chart. You must use the record and field name that you associated with the chart control on the page.

   ```
   &oChart = GetChart(QE_CHART_DUMREC.QE_CHART_FIELD);
   ```

9. Set the data records.

   Set the data record. The SetData function associates the record data with the chart. Then use the SetDataYAxis and SetDataXAxis functions to associate specific fields in the record with the Y axis data and the X axis data.

   ```
   &oChart.SetData(Record.QE_CHART_RECORD);
   &oChart.SetDataYAxis(QE_CHART_RECORD.QE_CHART_SALES);
   &oChart.SetDataXAxis(QE_CHART_RECORD.QE_CHART_PRODUCT);
   ```

   This is all the code needed to create a chart. You don't need to set the chart type: the default is a 2D bar chart. You don't need to set the series unless you want to group your data. Everything else is also optional.

10. (Optional) Set the data series.

    In this example, we want to set the region as the series. This means that the data will be grouped according to the region.

    ```
    &oChart.SetDataSeries(QE_CHART_RECORD.QE_CHART_REGION);
    ```

11. (Optional) Set the chart type.

    Because we want a stacked bar for the chart, we must set the Type property of the chart. This means that each product (footballs, rackets, and so on) will have a single bar, and the data series (California, Oregon, and so on) will be what is 'stacked'.

    ```
    &oChart.Type = %ChartType_2DStackedBar;
    ```

12. (Optional) Set legend and label attributes.

    For this example, we want a legend, and want it to appear in the right side. In addition, because the text of the series labels is so large, they must be vertical to display all of them.

    ```
    &oChart.HasLegend = True;
    &oChart.LegendPosition = %ChartLegend_Right;

    &oChart.XAxisLabelOrient = %ChartText_Vertical;
    ```

Creating a 'Gantt' Chart

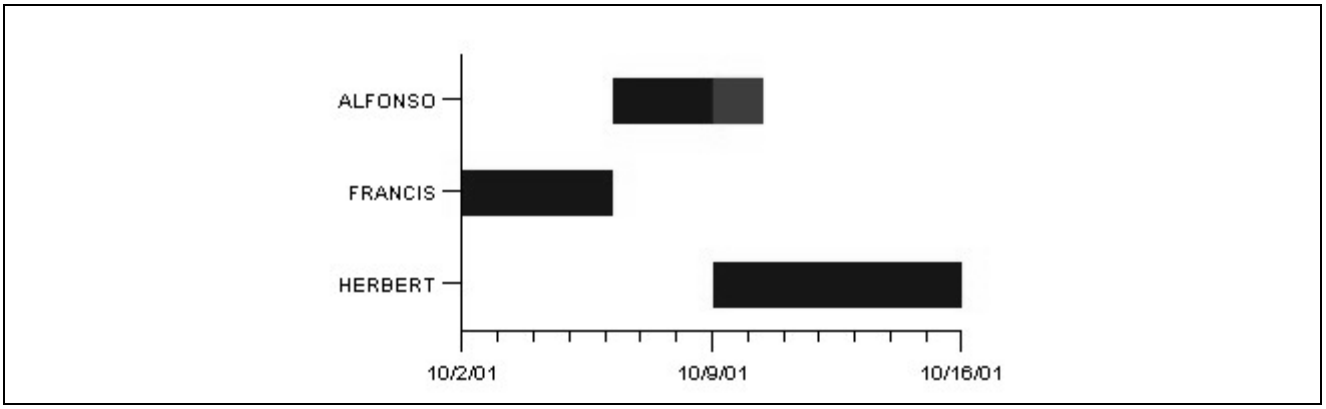
While the Chart Class may not support an actual Gantt chart type, you can create a chart that appears similar to a Gantt chart, by setting the Y axis labels on a horizontal stacked bar chart to the relevant data. The YAxisMax property must also be used to synchronize the end of this axis with the maximum number of stacked bars.

The following is the complete code sample: the steps explain each line.

```
Local Chart &oChart;

&oChart = GetChart(QE_CHART_DUMREC.QE_CHART_FIELD);
```

```
&oChart.Type = %ChartType_2DHorizStackedBar;

&oChart.SetData(Record.QE_CHART_RECORD);

&oChart.SetYAxisLabels(CreateArray("10/2/01", "10/9/01", "10/16/01"));

/* sets the number of ticks along the Y axis so there are seven intervals to⇒
 represent days */

&oChart.YAxisTicks = 6;

/* set the max of the Y axis so that the bars stretch to the end of the axis. In⇒
 this case the combine height of the highest stacked bar is 14. */

&oChart.YAxisMax = 14;


/* The Y data represents the starting and stopping points of the bars, that is,⇒
 for Alfonso (see chart, above) the value of the first series for point 'Alfonso'⇒
 along the X axis is 4 with the color of this bar set to white. The second series⇒
 is 3 with a blue color and the third series is 2, set to red. */

&oChart.SetDataYAxis(QE_CHART_RECORD.QE_CHART_WORK_SCHEDULE);

&oChart.SetDataXAxis(QE_CHART_RECORD.QE_CHART_EMPLOYEE);

/* Split the data into the different series to represent the bars. In this case⇒
 there are three. */

&oChart.SetDataSeries(QE_CHART_RECORD.QE_CHART_SERIES);

/* Set the colors of the bar so that they are on or off (white) */
&oChart.SetDataColor(QE_CHART_RECORD.QE_CHART_COLOR);
```

The following is the chart produced by the previous code:



Example of Gantt-like chart

To create a Gantt-like chart:

1. Get the chart, set the chart type and the chart data.

   Get the chart using the GetChart function. (You could also get the chart using the CreateObject function, as discussed in the following example.) To simulate the Gantt chart, set the type as a 2D horizontal stacked bar. Then set the record data for the chart.

   ```
   &oChart = GetChart(QE_CHART_DUMREC.QE_CHART_FIELD);

   &oChart.Type = %ChartType_2DHorizStackedBar;
   &oChart.SetData(Record.QE_CHART_RECORD);
   ```

2. Set the Y axis labels.

   Set the Y axis labels to the relevant data. In this example, it's set to a day in the week.

   ```
   &oChart.SetYAxisLabels(CreateArray("10/2/01", "10/9/01", "10/16/01"));
   ```

3. Set the Y axis ticks.

   Set the number of ticks along the Y axis so there are seven intervals to represent days.

   ```
   &oChart.YAxisTicks = 6;
   ```

4. Set the maximum of the Y axis value.

   You must set the maximum of the Y axis so that the bars stretch to the end of the axis. In this example, the combined height of the highest stacked bar is 14.

   ```
   &oChart.YAxisMax = 14;
   ```

5. Set the X and Y axis data.

   The Y data represents the starting and stopping points of the bars, that is, for Alfonso (see the previous chart) the value of the first series for point 'Alfonso' along the X axis is 4.

   ```
   &oChart.SetDataYAxis(QE_CHART_RECORD.QE_CHART_WORK_SCHEDULE);

   &oChart.SetDataXAxis(QE_CHART_RECORD.QE_CHART_EMPLOYEE);
   ```

6. Set the series.

   Split the data into the different series to represent the bars. In this case there are three.

   ```
   &oChart.SetDataSeries(QE_CHART_RECORD.QE_CHART_SERIES);
   ```

7. Set the colors.

   Set the colors of the bar so that they are on or off (white).

   ```
   &oChart.SetDataColor(QE_CHART_RECORD.QE_CHART_COLOR);
   ```

Creating a Chart Using an iScript

You can also create a chart using an iScript. The following example creates a chart, populates it, then sends the chart URL to a response object. The following is the complete code sample: the steps explain each line.

```
Function IScript_GetChartURL()

   Local Chart &oChart;
```

```
    Local Rowset &oRowset;
    Local string &sMap;
    Local string &sURL;

    &oChart = CreateObject("Chart");

    &oRowset = CreateRowset(Record.QE_CHART_RECORD);
    &oRowset.Fill("where QE_CHART_REGION= :1", "MIDWEST");
    &oChart.SetData(&oRowset);

    &oChart.Width = 400;
    &oChart.Height = 300;

    &oChart.SetDataYAxis(QE_CHART_RECORD.QE_CHART_SALES);
    &oChart.SetDataXAxis(QE_CHART_RECORD.QE_CHART_PRODUCT);
    &oChart.SetDataSeries(QE_CHART_RECORD.QE_CHART_REGION);

    &oChart.HasLegend = True;
    &oChart.LegendPosition = %ChartLegend_Right;

    &sURL = %Response.GetChartURL(&oChart);
    &sMap = &oChart.ImageMap;

    %Response.Write("<HTML><IMG SRC=");
    %Response.Write(&sURL);
    %Response.Write("  USEMAP=#THEMAP></IMG><MAP NAME=THEMAP>");
    %Response.Write(&sMap);
    %Response.Write("</MAP></HTML>");

End-Function;
```

To create a chart using an iScript:

1. Create the chart object.

   For this example, there is no chart control on a page to be referenced. So to create the chart object, you must use the CreateObject function. The string passed in to the function must be the word ’chart’.

   ```
   &oChart = CreateObject("Chart");
   ```

2. Create a rowset for the chart data and set the chart data.

   The CreateRowset function creates a rowset data structure. Using the Fill method populates the empty rowset with data. The SetData method associates the rowset data with the chart.

   ```
   &oRowset = CreateRowset(Record.QE_CHART_RECORD);
   &oRowset.Fill("where QE_CHART_REGION= :1", "MIDWEST");
   &oChart.SetData(&oRowset);
   ```

3. Set the height and width for the chart.

   Because the chart isn’t associated with a chart control on a page, you have to specify the size of the chart image to be generated using the Height and Width properties. The unit of measurement for both these properties is pixels.

```
&oChart.Width = 400;
 &oChart.Height = 300;
```

4. Set the data axes, series, and legend for the chart.

   As with all charts, you must set the data axes. If necessary for your data, also set the data series and the legend.

   ```
   &oChart.SetDataYAxis(QE_CHART_RECORD.QE_CHART_SALES);
   &oChart.SetDataXAxis(QE_CHART_RECORD.QE_CHART_PRODUCT);
   &oChart.SetDataSeries(QE_CHART_RECORD.QE_CHART_REGION);

   &oChart.HasLegend = True;
   &oChart.LegendPosition = %ChartLegend_Right;
   ```

5. Generate the URL for the chart.

   Use the GetChartURL method of the Response class to generate the URL referencing the chart data.

   ```
   &sURL = %Response.GetChartURL(&oChart);
   ```

6. Generate the image map.

   Use the ImageMap property to generate the image map for the chart. The Response object uses this to draw the map.

   ```
   &sMap = &oChart.ImageMap;
   ```

7. Create the response chart.

   Use the Write Response class method to generate the chart. Note that first the data is set, using the URL generated with GetChartURL, then the image map is used.

   ```
   %Response.Write("<HTML><IMG SRC=");
   %Response.Write(&sURL);
   %Response.Write("  USEMAP=#THEMAP></IMG><MAP NAME=THEMAP>");
   %Response.Write(&sMap);
   %Response.Write("</MAP></HTML>");
   ```

CHAPTER 10

Component Interface Classes

This chapter provides an overview of Component Interface class and discusses the following topics:

- Life-cycle of a Component Interface.
- Declaring a Component Interface object.
- Scope of a Component Interface object.
- Implementing a Component Interface object.
- Component Interface methods and timeouts.
- Traversing a Component Interface and using data collections.
- Working with effective-dated data.
- Reusing existing code.
- Component Interface Reference.

Understanding Component Interface Class

Component Interfaces are the focal points for externalizing access to existing PeopleSoft components. They provide realtime synchronous access to the PeopleSoft business rules and data associated with a component outside the PeopleSoft online system. Component Interfaces can be viewed as "black boxes" that encapsulate PeopleSoft data and business processes, and hide the details of the structure and implementation of the underlying page and data.

Component Interfaces are one of the many APIs that PeopleSoft provides for enabling integration with other systems.

A Component Interface maps to one, and only one, PeopleSoft component. The *Component Interface object*, instantiated from a session object, is created at runtime to access the data specified by the Component Interface.

When you instantiate a Component Interface object:

- All the PeopleCode programs associated with the record fields, pages, component, and so on, and
- The runtime component processor still perform all of the work that they do in the online environment.

The exceptions are any GUI manipulation found in a PeopleCode program, and search dialog specific processing.

Component Interfaces are programmable through a C interface, an OLE/COM or C/C++ interface, and through PeopleCode. Application Engine programs, application message subscription programs, or any other PeopleCode programs are able to use Component Interfaces.

Like a component, you create the structure of a Component Interface in PeopleSoft Application Designer, then at runtime, you populate the structure with data. This document is concerned with the runtime portion of a Component Interface.

When you populate a Component Interface with data, the first thing you fill out are its keys, as you would in a component. These can be keys for getting an existing instance of the data or for creating a new instance of the data.

In addition to keys, a Component Interface is composed of *properties* and *methods*.

- Component Interface *properties* provide access to the data in a component buffer.
- Component Interface *methods* are functions that can be called to perform operations on a Component Interface.

There are two types of both methods and properties: standard and user-defined. Standard properties and methods are provided automatically when you create a Component Interface. They perform operations common to all Component Interfaces, such as indicating what mode to operate the Component Interface, saving, or creating a Component Interface. User-defined properties are the specific record fields that an application developer has chosen to expose to an external system with the Component Interface. User-defined methods are PeopleCode programs that an application developer can write to perform operations on a Component Interface. Each is specific to that Component Interface.

You can instantiate a Component Interface object only from a session object. Through the session object you can control access to the Component Interface, check for errors, control the runtime environment, and so on.

See Also

Chapter 10, Component Interface Classes , Reusing Existing Code , page 783

Chapter 29, Session Class , page 1559

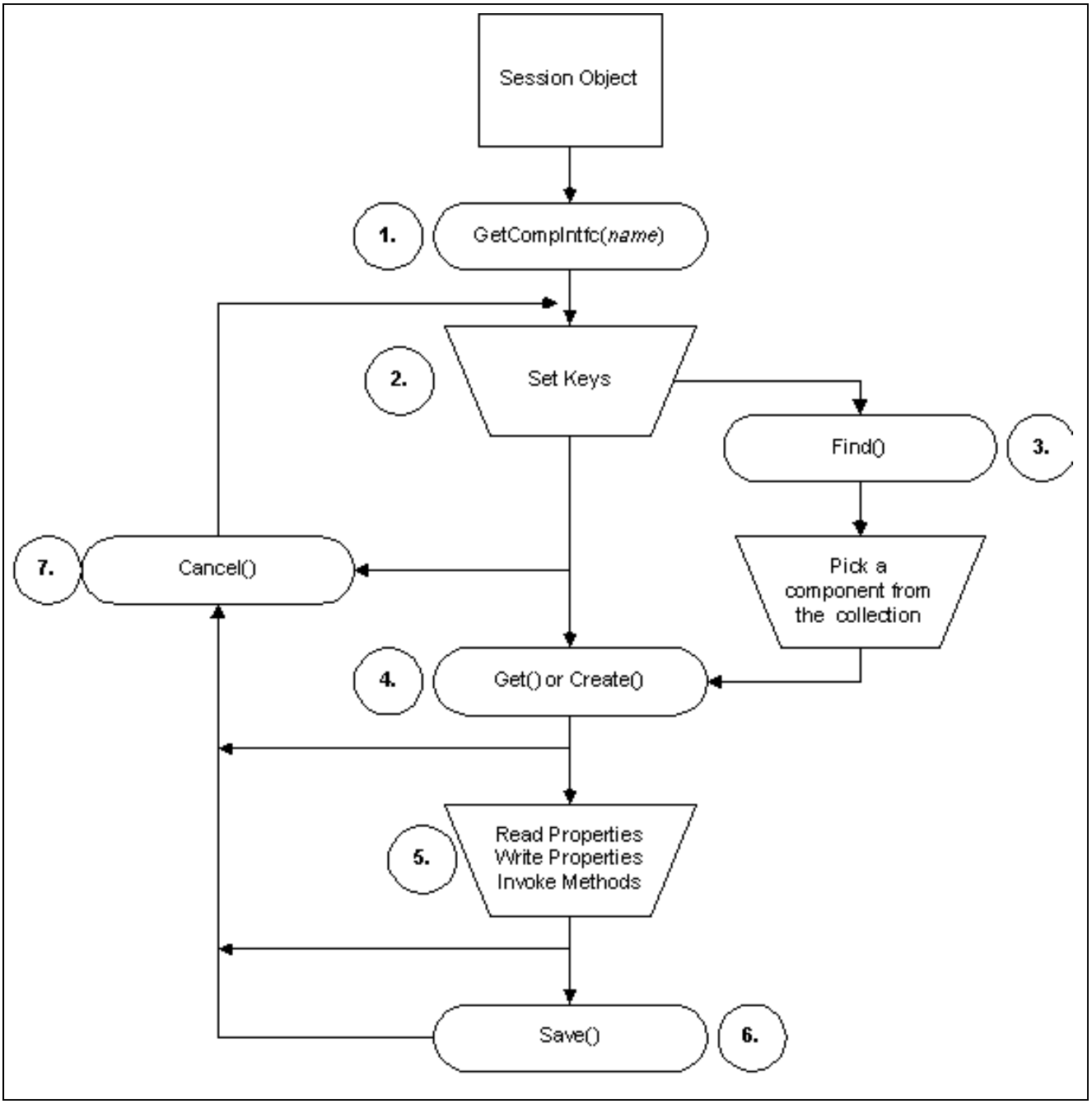
PeopleSoft Component Interfaces

Life Cycle of a Component Interface

At runtime, there are certain things you want to do with a Component Interface, such as getting an instance of it, populating it with data, and so on. The following is an overview of this process. These steps are expanded in other sections.

1. Execute the GetCompIntfc method on the PeopleSoft session object to get a Component Interface.
2. Set the key values for the Component Interface object. If the keys you specify don't uniquely describe a component (partial keys), proceed to step 3. If the keys uniquely describe a component, skip to step 4.
3. Do *one* of the following:
   - Execute the Find standard method on the Component Interface. This returns a collection of Component Interfaces with their key values filled out. The user can then select the unique Component Interface they want.
   - Execute either the Get or Create standard method to instantiate the Component Interface (populate it with data.)

4. Get property values, set property values, or execute user-defined methods. Setting a property value will fire the standard PeopleSoft business rules for the field associated with the property (any PeopleCode program associated with FieldChange, RowInsert, and so on.)
5. Execute the Save standard method to fire the standard PeopleSoft save business rules (any PeopleCode programs associated with SavePreChange, WorkFlow, and so on.) and commit any changes to the database.
6. At any point, the standard method Cancel can be executed to reset the Component Interface to its state in step 1.



Life cycle of a Component Interface

Setting Component Interface Keys

The keys for a Component Interface are based on the key fields of the underlying component. There can be different types of keys for a Component Interface.

- CREATEKEYS: A list of the primary key fields of the search record specified to be used in Add mode for the component. This list is automatically generated.
- GETKEYS: A list of the primary (required) key fields on the search record. This list is automatically generated.
- FINDKEYS: A list of primary and alternate key fields on the primary search record for the component.

If the Component Interface has CREATEKEYS, these are the keys you must set before you execute the Create() method to create a new instance of the data. If the Component Interface doesn't have CREATEKEYS, use the GETKEYS to specify a new instance of the data.

Use either the GETKEYS or FINDKEYS to specify an existing instance of the data.

To set key values, use the field names listed under GETKEYS, CREATEKEYS or FINDKEYS like properties on the Component Interface object. The following example sets the CREATEKEYS values to create a new instance of the data.

```
&MYCI = GetCompIntfc(CompIntfc.EXPRESS_ISSUE);
&MYCI.BUSINESS_UNIT = "H01B";
&MYCI.INTERNAL_FLG = "Y";
&MYCI.ORDER_NO = "NEXT';
&MYCI.Create();
```

Standard and User-Defined Component Interface Methods

Every Component Interface comes with a standard set of methods:

- Cancel()
- Create()
- Find()
- Get()
- Save()

Use these methods during runtime to affect the data of the Component Interface.

The application developer can, at design time, disable any of these methods for the Component Interface.

In addition, an application developer can write their own methods. These methods are written as Functions using Component Interface PeopleCode. For example, suppose you wanted to be able to copy an instance of Component Interface data. You might write your own Clone method.

Note. User-defined method names must *not* be named **Get** *PropertyName*. The C header for Component Interfaces creates functions with that name so you can access each property. If you create your own **Get** *PropertyName* functions, you receive errors at runtime. User-defined methods can take only simple types of arguments (such as number, character, and so on) because user-defined methods can be called from C/C++ or COM as well as from PeopleCode. PeopleCode can use more complex types (like rowset, array, record, and so on), but these types of arguments are unknown to C/C++ or COM.

See Also

Chapter 10, Component Interface Classes , Component Interface Class Methods , page 787

PeopleSoft Component Interfaces

PeopleSoft Component Interfaces, Developing the Component Interface, Creating User-Defined Methods

Accessing Component Interface Standard Properties

Every Component Interface comes with a standard set of properties. These properties can be divided as follows:

Properties that affect how the Component Interface is executed

The following properties affect how a component interface is executed:

- EditHistoryItems
- GetHistoryItems
- InteractiveMode

These properties must be set before the Component Interface is populated with data. That is, you must set these properties before you use the **Get** or **Create** methods.

EditHistoryItems and GetHistoryItems work together to determine how data is accessed:

- If EditHistoryItems is False (the default) and GetHistoryItems is True, you access the data in the Component Interface in a similar manner as if you were accessing a component in update/display All mode. This means all history rows are returned, however, you can edit rows only with a date set in the future.
- If EditHistoryItems is True and GetHistoryItems is True, you access the data in the Component Interface in a similar manner as if you were accessing a component in correction mode. This means all history rows are returned, and you can edit them.
- If GetHistoryItems is False, you access the data in the Component Interface in a similar manner as if you were accessing a component in update mode. The EditHistoryItems has no effect when GetHistoryItems is False.

  InteractiveMode causes the Component Interface to emulate an online component. For example, if you set a value for a field in a Component Interface and you have set InteractiveMode to True, then any FieldChange PeopleCode programs associated with that field fire as soon as you set that value.

Properties that return information about the structure of the Component Interface

The following properties return information about the structure of the Component Interface:

- CreateKeyInfoCollection
- FindKeyInfoCollection
- GetKeyInfoCollection
- PropertyInfoCollection

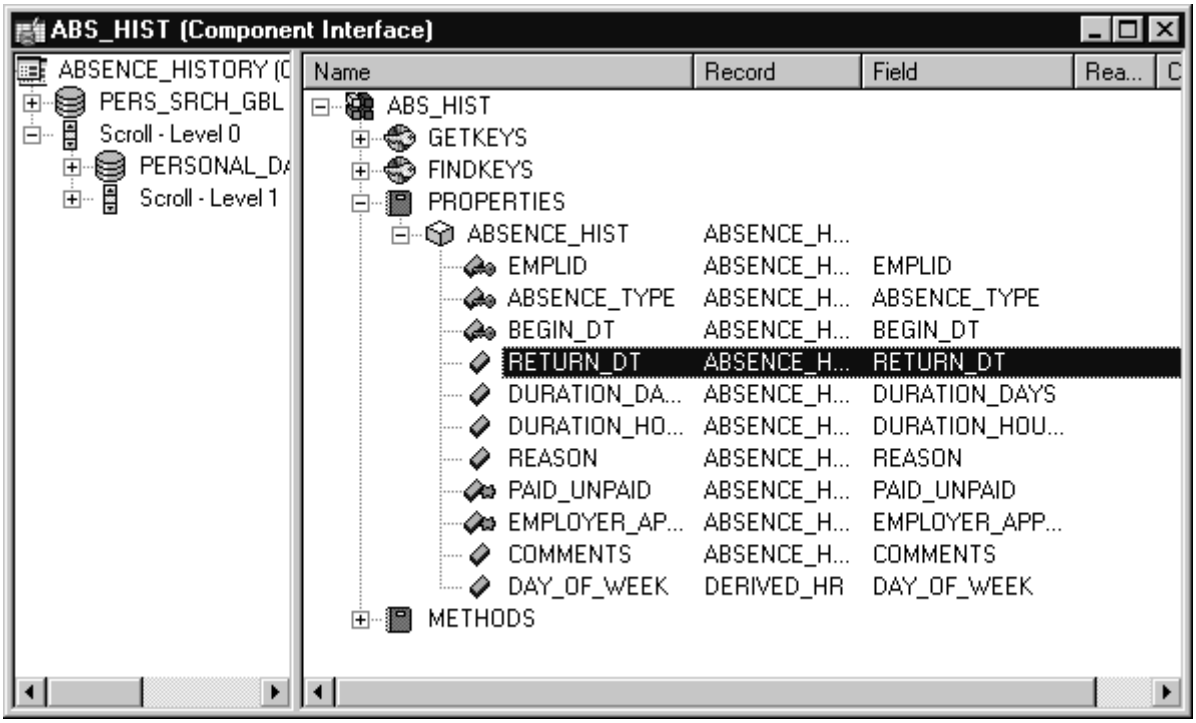
See Also

Chapter 10, Component Interface Classes , Component Interface Class Properties , page 801

Accessing User-Defined Component Interface Properties

Every user-defined property in a Component Interface *definition* can be used like a property on the *object* instantiated from that Component Interface at runtime.

For example, the following Component Interface definition has RETURN_DT defined as one of its properties.



RETURN_DT Component Interface property highlighted

At runtime, you can use PeopleCode to assign a value to that property (field), or to access the value of that field.

```
&MYCI.RETURN_DT = "05/05/2000";

/*  OR  */

&DATE = &MYCI.RETURN_DT;
```

Data Type of a Component Interface Object

Component Interfaces are declared as type ApiObject. For example,

```
Local ApiObject &MYCI;
```

Note. Component Interface objects can be declared only as Local.

Scope of a Component Interface Object

A Component Interface can be instantiated from PeopleCode, from a Visual Basic program, from Java, COM and C/C++.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

If you're instantiating a Component Interface from an online page, after you finish working with the component, you may want to refresh your page. The **Refresh** method, on a rowset object, reloads the rowset (scroll) using the current page keys. This causes the page to be redrawn. GetLevel0().Refresh() refreshes the entire page. If you want a particular scroll only to be redrawn, you can refresh just that part.

Considerations Using Component Interfaces and Related Languages

If you update a field using a Component Interface through an external program (such as Java, VB, and so on) the Related Language Table gets updated, even if they are using a base language of ENG.

Considerations Using Component Variables With Component Interfaces

A component variable is scoped locally to its component or its Component Interface. This means that you cannot use a component variable to share data between a component and a Component Interface. In order to share information between components, use global variables.

If your page (component) calls a Component Interface (using the existing session), that in turn initializes a component variable, that component variable is shared with the calling component. When the Component Interface is closed, the component variable is no longer in scope.

When the calling component is closed, any component scoped variables created by it or by the Component Interface go out of scope.

See Also

Chapter 27, Rowset Class , Refresh, page 1495

PeopleSoft Component Interfaces

Implementing a Component Interface

After you create your Component Interface definition, you can use PeopleCode to access it. This PeopleCode can be long and complex. Rather than write it directly, you can drag and drop the Component Interface definition from PeopleSoft Application Designer Project View into an open PeopleCode edit pane. PeopleSoft Application Designer analyzes the definition and generates initial PeopleCode as a template, which you can modify.

You can also access your Component Interface using COM. You can automatically generate a Visual Basic or a C template, similar to the PeopleCode template, to get you started.

The following are the usual actions that you perform with a Component Interface:

- Create a new instance of data

- Get an existing instance of data
- Retrieve a list of instances of data

The following procedures cover each of these actions in more detail.

Another standard action is inserting or deleting a row of data (an item). This involves traversing a Component Interface (going from level 0 to level 1, from level 1 to level 2, and so on) and accessing data collections.

You may want to work with effective-dated information. There are several properties you can set to allow you to do this.

In addition to these standard actions, you can also look at the structure of a Component Interface.

See Also

Chapter 10, Component Interface Classes , Create a New Instance of Data Example, page 820

Chapter 10, Component Interface Classes , Getting an Existing Instance of Data Example, page 822

Chapter 10, Component Interface Classes , Retrieving a List of Instance of Data Example, page 825

Chapter 10, Component Interface Classes , Traversing a Component Interface and Using Data Collections , page 780

Chapter 10, Component Interface Classes , Working With Effective-Dated Data , page 782

Chapter 10, Component Interface Classes , Accessing the Structure of a Component Interface , page 812

Component Interface Methods and Timeouts

The file pstools.properties controls when a Component Interface method times out. If you're having problems with methods timing out, you may want to change the values in this file. This file is located in the default directory for the application you're running. If this file isn't in this directory, copy it into the directory before you make your changes to it.

Traversing a Component Interface and Using Data Collections

The data in a Component Interface can be contained in a hierarchy: like the page it's built on, there may be data at level 0, level 1, level 2, and so on.

Each level of data in a Component Interface is known as a collection, such as:

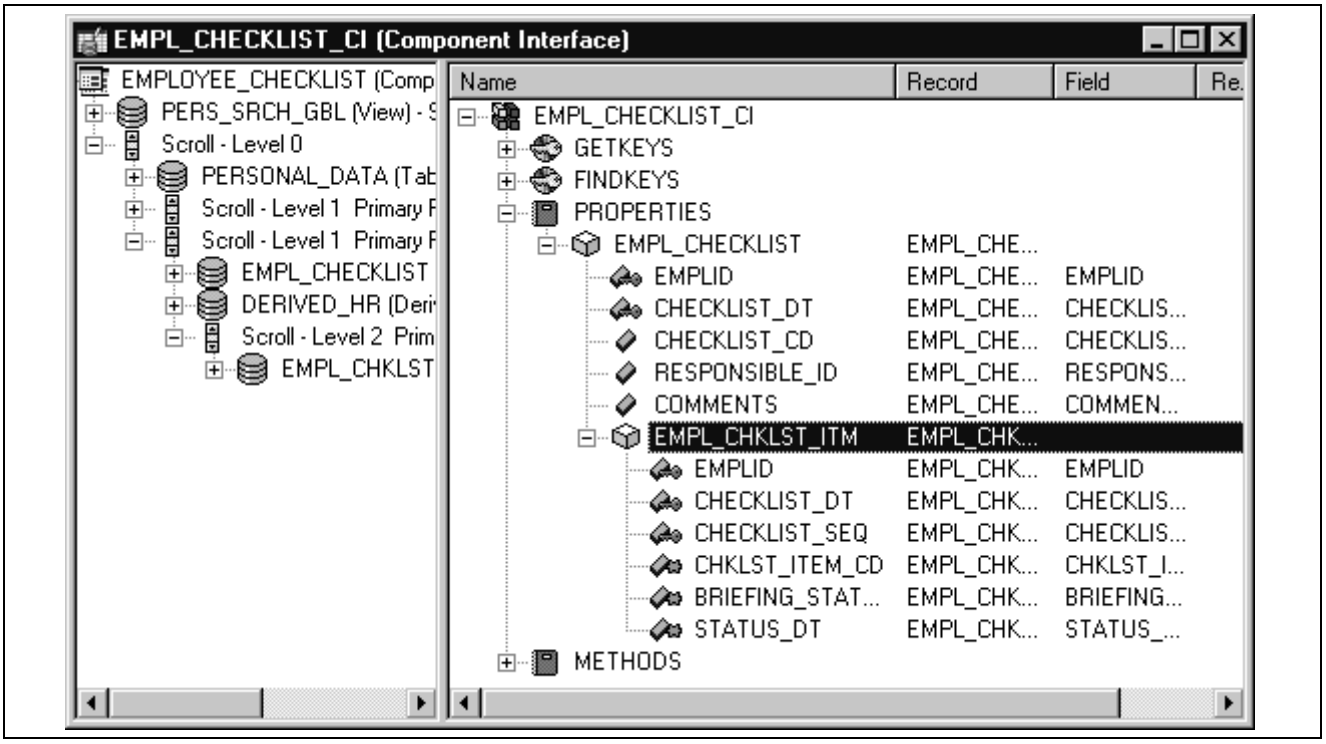
```
Level 0

  -- Level 1 (Collection)

   -- Level 2 (Collection)
```

A *collection* is a set of similar things, like a group of already existing Component Interfaces. Most collections have the same standard properties and methods, with some additional ones specific to that collection. For example, all collections have the property *Count*, which indicates how many items are in that collection, but only a data collection has the method *ItemByKeys*.

A *data collection* is the collection of data, available at runtime or during test mode, associated with a particular scroll (or record.) The data collection object returns information about every *row of data* (item) that is returned for that record at runtime.

To access the level 2 collection, in general, you could use the following:

```
&Level_1 = &CI.Level_1;
&Level_1_Item = &Level_1.Item(ItemNumber);

&Level_2 = &Level_1_Item.Level_2;
```



Sample Component Interface with collection highlighted

This example shows a Component Interface with a two-level hierarchy, that is, *two* data collections: EMPL_CHECKLIST and EMPL_CHKLST_ITM. To get a data collection for the first level, use the following code:

```
&Level1 = &MYCI.EMPL_CHECKLIST;
```

To access the secondary scroll (EMPL_CHKLST_ITM) you have to get the appropriate row (item) of the upper level scroll first:

```
&Level1 = &MYCI.EMPL_CHECKLIST;
&Item = &Level1.Item(1);
&Level2 = &Item.EMPL_CHKLST_ITM;
```

Note. Scrolls represent a hierarchical order. You must get the first level row that contains the secondary scroll *before* you can access the secondary scroll.

Every data collection has a set of methods and standard properties.

See Also

Chapter 10, Component Interface Classes , Inserting or Deleting a Row of Data Example, page 831

Chapter 10, Component Interface Classes , Data Collection Methods , page 805

Chapter 10, Component Interface Classes , Data Collection Properties , page 811

PeopleCode Developer's Guide, Data Buffer Access

Working With Effective-Dated Data

When you work with effective-dated data, use some combination of the following methods:

- CurrentItem
- GetEffectiveItem

Both these methods return an item, however, there are some differences:

- The CurrentItem method takes no parameters, so it bases the item it returns on the *current system date.*
- The GetEffectiveItem method bases the item it returns on a *user-provided date*. They all perform the same logic, they just use a different date.

Note. To show all history for an effective-dated collection, you must set GetHistoryItems to **True** before you populate the Component Interface.

See Also

Chapter 10, Component Interface Classes , CurrentItem, page 805

Chapter 10, Component Interface Classes , GetEffectiveItem, page 807

Chapter 10, Component Interface Classes , GetHistoryItems, page 803

Chapter 10, Component Interface Classes , Inserting Effective-Dated Data Example, page 827

Chapter 10, Component Interface Classes , Inserting Effective-Dated Data Example Using Visual Basic, page 829

How a Developer Uses GetEffectiveItem

If a user is making an update to an effective-dated record, they don't always want to insert a row at the end.

Suppose the database contained the following data:

EMPLID	EFFDT	SEQNO
8000	3/1/99	0
8000	7/1/99	0
8000	9/1/99	0
8000	12/1/99	0

Now suppose the user wants to enter info with EFFDT of 11/1/99. If they were looking at a PeopleSoft component, they would visually scan to see where that date falls and then press ALT+7 and ENTER at the row that they want to insert after.

The GetEffectiveItem enables you to pass in the correct effective date, instead of having to loop through every item in the collection doing a comparison, looking for the correct item.

Why Can't it Just Go at the End?

The InsertItem method simulates pressing ALT+7 and ENTER online to insert a row in a scroll. Part of the logic that occurs in the Component Processor is that if the scroll is effective-dated, the ALT+7 and ENTER carries the values forward from the previous row. This functionality is still there if you use InsertItem() at the end of the collection, but the values may be incorrect.

Reusing Existing Code

One of the advantages of using a Component Interface is that it enables you to reuse existing PeopleCode and business logic. However, a Component Interface isn't exactly equivalent to a component, which means there are a few key areas in which you should expect differences in behavior between a Component Interface and the component it's based on. This section discusses the differences in:

- Search dialog processing
- Event and function behavior

Differences in Search Dialog Processing

When you run a Component Interface, the *SearchInit*, *SearchSave*, and *RowSelect* events don't fire. This means that any PeopleCode associated with these events won't run. The first event to run is *RowInit*.

Differences in PeopleCode Event and Function Behavior

PeopleCode events and functions that relate exclusively to GUI and online processing can't be used by Component Interfaces. These include:

- Menu PeopleCode and pop-up menus. The **ItemSelected** and **PrePopup** PeopleCode events are not supported. In addition, the **DisableMenuItem**, **EnableMenuItem**, and **HideMenuItem** functions aren't supported.
- Transfers between components, including modal transfers. The **DoModal, EndModal, IsModal, Transfer, TransferPage, DoModalComponent, TransferNode, TransferPortal,** and **IsModalComponent** functions cannot be used.
- Cursor position. **SetControlValue** cannot be used.
- **WinMessage** cannot be used.
- Save in the middle of a transaction. **DoSave** can't be used.

For the unsupported functions, you should put a condition around them, testing whether there's an existing Component Interface.

```
If %ComponentName Then
   /* process is being called from a Component Interface */
   /* do CI specific processing */
Else
   /* do regular processing */
   .  .  .
End-if;
```

Using %Menu Conditions

If you associate a Component Interface definition with a menu in Application Designer, a PeopleCode program that conditionally checks for that menu runs when the Component Interface is executed. For example, suppose you associate a Component Interface with the MP_HR_MENU. The following PeopleCode program, if called from that Component Interface, executes:

```
If %Menu = MP_HR_MENU Then
```

See Also

PeopleSoft Component Interfaces, Developing the Component Interface, Associating a Component Interface With a Menu

Session Class Methods

Component Interfaces don't have any built-in functions. They are instantiated from the Session class.

This section discusses the Session class methods used to instantiate Component Interfaces. The methods are discussed in alphabetical order.

FindCompIntfcs

Syntax

FindCompIntfcs([*ComponentName*])

Description

The **FindCompIntfcs** method, used with the session object, returns a reference to a Component Interface collection, filled with zero or more Component Interfaces. The *ComponentName* parameter takes a string value. You can use a partial value to limit the set of Component Interfaces returned. You can also specify a null value, that is, two quotation marks with no space between them, ("") to return the entire list of Component Interfaces available.

Using the **FindCompIntfcs** method is equivalent to using File, Open, and selecting Component Interface in PeopleSoft Application Designer.

Example

In the following example, a partial key was used to open all components starting with "APP".

```
Local ApiObject &MYSESSION;
Local ApiObject &MYCI;

&MYSESSION = %Session;
&MYCICOLL = &MYSESSION.FindCompIntfcs("APP");
&MYCI = &MYCICOLL.First();
```

GetCompIntfc

Syntax

GetCompIntfc([**COMPINTFC**.]*ComponentName*)

Description

The **GetCompIntfc** method, used with the session object, returns a reference to a Component Interface. *ComponentName* when used by itself takes a string value. If you use COMPINTFC.*componentname* it isn't a string value: it's a constant that automatically is renamed in your code if the Component Interface definition is ever renamed. You must specify an existing Component Interface, otherwise you receive a runtime error.

Example

```
Local ApiObject &MYSESSION;
Local ApiObject &MYCI;

&MYSESSION = %Session;
&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.PERSONAL_DATA_BC);
```

The following example uses the '@' operator to dynamically call a Component Interface from PeopleCode.

```
&MYCI = &MYSESSION.GetCompIntfc(@("CompIntfc." | &CIname));
```

Component Interface Collection Methods

A Component Interface collection is a list of the available Component Interfaces. An equivalent list is generated by starting PeopleSoft Application Designer, selecting File, Open, Component Interface.

You get a Component Interface collection by using the FindCompIntfcs method with a session object.

First

Syntax

First()

Description

The **First** method returns the first Component Interface in the Component Interface collection object executing the method. This returns the structure of the Component Interface with only the key fields filled in. The rest of the data is not present. To populate the Component Interface with data, you must set the key values and use the **Get** method.

Example

```
&MYCI = &CICOLLECTION.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** returns the Component Interface that exists at the *number* position in the Component Interface collection executing the method. This returns the structure of the Component Interface with only the key fields filled in. The rest of the data is not present. To populate the Component Interface with data, you must set the key values and use the **Get** method. *number* takes a number value.

Example

```
For &I = 1 to &COLLECTION.Count;
&MYCI = &COLLECTION.Item(&I);
/* do processing */
End-For;
```

Next

Syntax

Next()

Description

The **Next** method returns the next Component Interface in the Component Interface collection object executing the method. You can use this method only after you have used either the **First** or **Item** methods: otherwise the system doesn't know where to start. This returns the structure of the Component Interface with only the key fields filled in. To populate the Component Interface with data, you must set the key values and use the **Get** method.

Example

```
&MYCI = &COLLECTION.Next();
```

Component Interface Collection Property

This section discusses the Count property.

Count

Description

This property returns the number of Component Interfaces in the Component Interface collection, as a number.

This property is read-only.

Example

```
&COUNT = &MYCI_COLLECTION.Count;
```

Component Interface Class Methods

This section discusses the Component Interface class methods. The methods are discussed in alphabetical order.

Cancel

Syntax

Cancel()

Description

The **Cancel** method cancels the instance of the Component Interface object executing the method, rolling back any changes that were made. This sets the Component Interface state to the same state it was in immediately after it was created by **GetCompIntfc**. This closes the Component Interface.

Parameters

None.

Returns

A Boolean value: True if component was successfully cancelled, False otherwise.

CopyRowset

Syntax

CopyRowset(*&Rowset* [, *InitialRow*] [, *record_list*]);

Where *record_list* is a list of record names in the form:

```
[RECORD.source_recname1, RECORD.target_recname1
[, RECORD.source_recname2, RECORD.target_recname2]].  .  .
```

Description

The **CopyRowset** method copies the specified rowset object to the Component Interface executing the method, copying *like-named* record fields and data collections (child rowsets) at corresponding levels. If pairs of source and destination record names are given, these are used to pair up the records and data collections *before* checking for like-named record fields.

CopyRowset uses the Page Field order when copying properties. This helps ensure that dependent fields are set in the required order.

Note. This method works in PeopleCode only. This method uses the names of the records in the collection, *not* the name you may give the collection when you create the Component Interface.

If there are blanks in *source* rowset or record, they are copied over to the Component Interface only if the field's **IsChanged** property is set to True. Otherwise blanks are *not* copied.

Use this method when you are using a Component Interface to verify the data in your application message.

Note. The structure of the rowset you're copying data from must exactly match the existing rowset structure, with the same records at level 0, 1, 2, and so on. **CopyRowset** is intended to be used with a subscription process, that is, with message data. As all subscriptions work only in two-tier mode, **CopyRowset** works only in two-tier mode.

Parameters

Parameter	Description
&Rowset	Specify an existing, instantiated rowset object that contains data.
InitialRow	Specify the initial transaction row to begin with. This parameter provides a quick way to loop through a rowset that has multiple level 0 rows. The default value for this parameter is 1.
record_list	Specify a list of source and target record names. All *source_recnames* are records being copied *from*, in the rowset object being copied from. All *target_recnames* are records being copied *to*, in the Component Interface being copied to.

Returns

None.

Example

The following example would be in your Subscription PeopleCode. The Exit(1) causes all changes to be rolled back, and the message is marked with the status ERROR so you can correct it.

```
Local message &MSG;
Local ApiObject &SESSION;
Local ApiObject &PO;
Local rowset &ROWSET;
```

```
&MSG = GetMessage();
&ROWSET = &MSG.GetRowset();

&SESSION = %Session;

If &SESSION <> NULL Then
/* Session connected correctly */
/* Set key values to create component */
&PO = &SESSION.GetCompIntfc(COMPINTFC.PO);
&PO.BU = &MSG(&I).PO_HDR.BU.Value;
&PO.PO_ID = &MSG(&I).PO_HDR.PO_ID.Value;
&PO.Create();

&PO.CopyRowset(&ROWSET);
If NOT (&PO.Save()) Then
   Exit(1);
End-if;
Else
/* do error processing */
End-If;
```

The following example loops through all the transactions of a message rowset.

```
Local message &MSG;
Local ApiObject &SESSION;
Local ApiObject &CI;
Local rowset &ROWSET;

&MSG = GetMessage();
&ROWSET = &MSG.GetRowset();


&SESSION = %Session;
If &SESSION <> Null Then)
   &CI = &SESSION.GetCompIntfc(CompIntfc.VOL_ORG);
   &I = 0;
   While (&I < &ROWSET.RowCount)
      &I = &I + 1;
      &CI.VOLUNTEER_ORG = &ROWSET.GetRow(&I).VOLNTER_ORG_2.  VOLUNTEER_ORG.Value;
      If &CI.Create() Then
         If &CI.CopyRowset(&ROWSET, &I, Record.VOLNTER_ORG_TBL, ⇒
Record.VOLNTER_ORG_TBL) Then

         /* App specific processing */

            If Not &CI.Save() Then
               Winmessage("Save Failed");
               /* Other app specific processing */
            End-If;
```

```
            End-If;
         End-If;
      &CI.Cancel();
      End-While;
   End-If;
```

CopyRowsetDelta

Syntax

CopyRowsetDelta(*&Rowset* [, *InitialRow*] [, *record_list*]);

Where *record_list* is a list of record names in the form:
`[RECORD.source_recname1, RECORD.target_recname1`
`[, RECORD.source_recname2, RECORD.target_recname2]]. . .`

Description

The **CopyRowsetDelta** method copies the *changed* rows in the specified rowset object to the Component Interface executing the method, copying *like-named* record fields and data collections (child rowsets) at corresponding levels. If pairs of source and destination record names are given, these are used to pair up the records and data collections *before* checking for like-named record fields.

Note. This method works in PeopleCode only.This method uses the names of the records in the collection, *not* the name you give the collection when you create the Component Interface.

If there are blanks in *source* rowset or record, they are copied over to the Component Interface only if the field's **IsChanged** property is set to True. Otherwise blanks are *not*copied.

You will generally use this method with a Component Interface to verify data from a message.

Note. CopyRowsetDelta is intended for a subscription process, that is, with message data. As all subscriptions work only in two-tier mode, **CopyRowsetDelta** works only in two-tier mode.

If the rowset you're copying from is a message rowset, the **CopyRowsetDelta** method uses the AUDIT_ACTN field in the PSCAMA table in the message to know whether the row is to be inserted, updated, or deleted inside the Component Interface.

If the rowset you're copying from *is not* a message rowset, that rowset must have the same structure as a message, that is, it must have a PSCAMA record with an AUDIT_ACTN field on every level of the rowset.

Warning! CopyRowsetDelta uses a record's keys to locate the target row to change for all audit actions other than **Add**. CopyRowsetDelta actions (other than Add) therefore work only on rowsets that have keys *that uniquely identify all rows in the rowset*. Rowsets that do *not*distinguish between rows using a key field will be updated in an unpredictable fashion.

Parameters

Parameter	Description
&Rowset	Specify an existing, instantiated rowset object that contains data.
InitialRow	Specify the initial transaction row to begin with. This parameter provides a quick way to loop through a rowset that has multiple level 0 rows. The default value for this parameter is 1.
record_list	Specify a list of source and target record names. All *source_recnames* are records being copied *from*, in the rowset object being copied from. All *target_recnames* are records being copied *to*, in the Component Interface being copied to.

Returns

None.

Example

The following PeopleCode would exist in your subscription process. The Exit(1) causes all changes to be rolled back, and the message is marked with the status ERROR so you can correct it.

```
Local Message &MSG;
Local ApiObject &SESSION;
Local ApiObject &PO;
Local Rowset &ROWSET;

&MSG = GetMessage();
&ROWSET = &MSG.GetRowset();

&SESSION = %Session;

If &SESSION <> Null Then
/* Session connected correctly */
/* Set key values to get component */
&PO = &SESSION.GetCompIntfc(COMPINTFC.PO);
&PO.BU = &MSG(&I).PO_HDR.BU.Value;
&PO.PO_ID = &MSG(&I).PO_HDR.PO_ID.Value;
&PO.Get();


&PO.CopyRowsetDelta(&ROWSET);
If NOT (&PO.Save()) Then
   Exit(1);
End-if;
Else
   /* do error processing */
End-If;
```

See Also

PeopleSoft Integration Broker, Sending and Receiving Messages, Understanding PSCAMA.

CopySetupRowset

Syntax

```
CopySetupRowset(&Rowset [, InitialRow] [, record_list]);
```

Where *record_list* is a list of record names in the form:
`RECORD.source_recname, RECORD.target_recname`

Description

The **CopySetupRowset** method is used to copy a setup table application message to a Component Interface. A setup table has the same record at level 0 and level 1, while a Component Interface has only a single copy of a record. This method copies the contents of the message (at level 0) to the first level collection (level 1) of the Component Interface.

The **CopySetupRowset** method copies *like-named* record fields. If a pair of source and destination record names are given, these are used to pair up the records and data collections *before* checking for like-named record fields.

Note. This method works in PeopleCode only. This method uses the names of the records in the collection, *not* the name you may give the collection when you create the Component Interface.

If there are blanks in *source* rowset or record, they are copied over to the Component Interface only if the field's **IsChanged** property is set to True. Otherwise blanks are *not* copied.

Note. CopySetupRowset is for a subscription process, that is, with message data. As all subscriptions work only in two-tier mode, **CopySetupRowset** works only in two-tier mode.

Parameters

Parameter	Description
&Rowset	Specify an existing, instantiated rowset object that contains data.
InitialRow	Specify the initial transaction row to begin with. This parameter provides a quick way to loop through a rowset that has multiple level 0 rows. The default value for this parameter is 1.
record_list	Specify source and target record names. The *source_recname* is the record being copied *from*, in the rowset object being copied from. The *target_recname* is the record being copied *to*, in the Component Interface being copied to.

Returns

None.

Example

The following example would be in your Subscription PeopleCode.

```
Local ApiObject &SESSION;
Local ApiObject &PSMESSAGES;

Local ApiObject &CI;
```

```
Local Message &MSG;
Local Rowset &RS;

&SESSION = %Session;
&PSMESSAGES = &SESSION.psmessages;

&MSG = GetMessage();
&RS = &MSG.GetRowset();

&CI = &SESSION.getcomponent(Component.VOL);

/** Set Business Component Properties **/
&CI.gethistoryitems = True;
/*&CI.InteractiveMode = True;*/
/*&CI.stoponfirsterror = True;*/

For &TRANSACTION = 1 To &RS.RowCount

   &CI.VOLUNTEER_ORG = &RS(&TRANSACTION).VOLNTER_ORG_TBL.VOLUNTEER_ORG.Value;

   /* note: You will achieve much better performance if you add code here to check⇒
  if the keys are the same L0 keys as the last time through the loop and, if so,⇒
 skip the Get/Create section. */

   If Not &CI.create() Then
      &PSMESSAGES.DeleteAll();
      If Not &CI.get() Then
         /** Check Error Messages **/
         For &I = 1 To &PSMESSAGES.count
            &TYPE = &PSMESSAGES.item(&I).type;
            &TEXT = &PSMESSAGES.item(&I).text;
         End-For;
         Exit (1);

      End-If;
   End-If;

   If Not &CI.CopySetupRowset(&RS, &TRANSACTION) Then
      /** Check Error Messages **/
      For &I = 1 To &PSMESSAGES.count
         &TYPE = &PSMESSAGES.item(&I).type;
         &TEXT = &PSMESSAGES.item(&I).text;
      End-For;
      Exit (1);
   End-If;

   If Not &CI.Save() Then
      /** Check Error Messages **/
      For &I = 1 To &PSMESSAGES.count
         &TYPE = &PSMESSAGES.item(&I).type;
```

```
            &TEXT = &PSMESSAGES.item(&I).text;
         End-For;
         Exit (1);
      End-If;

      &CI.Cancel();
   End-For;
```

See Also

Chapter 10, Component Interface Classes , CopySetupRowsetDelta, page 794, Chapter 10, Component Interface Classes , CopyRowset, page 787.

CopySetupRowsetDelta

Syntax

CopySetupRowsetDelta(*&Rowset* [, *InitialRow*] [, *record_list*]);

Where *record_list* is a list of record names in the form:
RECORD.*source_recname*, RECORD.*target_recname*

Description

The **CopySetupRowsetDelta** method is used to copy a setup table with *changed* rows in an application message to a Component Interface. A setup table has the same record at level 0 and level 1, while a Component Interface has only a single copy of a record. This method copies the contents of the message (at level 0) to the first level collection (level 1) of the Component Interface.

Note. This method works in PeopleCode only.CopySetupRowsetDelta copies all the like-named fields from the changed *row* into the message. It is *not* copying only the changed fields.

The **CopySetupRowsetDelta** method copies *like-named* record fields, in those rows where the **IsChanged** property is True. If a pair of source and destination record names are given, these are used to pair up the records and data collections *before* checking for like-named record fields.

Note. This method uses the names of the records in the collection, *not*the name you may give the collection when you create the Component Interface.

If there are blanks in *source* rowset or record, they are copied over to the Component Interface only if the field's **IsChanged** property is set to True. Otherwise blanks are *not*copied.

Note. CopySetupRowsetDelta is for a subscription process, that is, with message data. As all subscriptions work only in two-tier mode, **CopySetupRowsetDelta** works only in two-tier mode.

Warning! CopySetupRowsetDelta uses a record's keys to locate the target row to change for all audit actions other than **Add**. CopySetupRowsetDelta actions (other than Add) therefore work only on rowsets that have keys that *uniquely identify all rows in the rowset.* Rowsets that do *not*distinguish between rows using a key field will be updated in an unpredictable fashion.

Parameters

Parameter	Description
&Rowset	Specify an existing, instantiated rowset object that contains data.
InitialRow	Specify the initial transaction row to begin with. This parameter provides a quick way to loop through a rowset that has multiple level 0 rows. The default value for this parameter is 1.
record_list	Specify source and target record names. The *source_recname* is the record being copied *from*, in the rowset object being copied from. The *target_recname* is the record being copied *to*, in the Component Interface being copied to.

Returns

None.

Example

```
Local ApiObject &SESSION;
Local ApiObject &PSMESSAGES;

Local ApiObject &CI;

Local Message &MSG;
Local Rowset &RS;

&SESSION = %Session;
&PSMESSAGES = &SESSION.psmessages;

&MSG = GetMessage();
&RS = &MSG.GetRowset();

&CI = &SESSION.getcomponent(Component.VOL);

/** Set Business Component Properties **/
&CI.gethistoryitems = True;
/*&CI.InteractiveMode = True;*/
/*&CI.stoponfirsterror = True;*/

For &TRANSACTION = 1 To &RS.RowCount

   &CI.VOLUNTEER_ORG = &RS(&TRANSACTION).VOLNTER_ORG_TBL.VOLUNTEER_ORG.Value;

   /* note: You will achieve much better performance if you add code here to check⇒
  if the keys are the same L0 keys as the last time through the loop and, if so,⇒
 skip the Get/Create section. */

   If Not &CI.create() Then
      &PSMESSAGES.DeleteAll();
      If Not &CI.get() Then
```

```
      /** Check Error Messages **/
      For &I = 1 To &PSMESSAGES.count
         &TYPE = &PSMESSAGES.item(&I).type;
         &TEXT = &PSMESSAGES.item(&I).text;
      End-For;
      Exit (1);

   End-If;
End-If;

If Not &CI.CopySetupRowsetDelta(&RS, &TRANSACTION) Then
   /** Check Error Messages **/
   For &I = 1 To &PSMESSAGES.count
      &TYPE = &PSMESSAGES.item(&I).type;
      &TEXT = &PSMESSAGES.item(&I).text;
   End-For;
   Exit (1);
End-If;

If Not &CI.Save() Then
   /** Check Error Messages **/
   For &I = 1 To &PSMESSAGES.count
      &TYPE = &PSMESSAGES.item(&I).type;
      &TEXT = &PSMESSAGES.item(&I).text;
   End-For;
   Exit (1);
End-If;

&CI.Cancel();
```

See Also

Chapter 10, Component Interface Classes , CopySetupRowset, page 792, Chapter 10, Component Interface Classes , CopyRowsetDelta, page 790.

Create

Syntax

Create()

Description

The **Create** method associates the Component Interface object executing the method with a new, open Component Interface that matches the key values that were set prior to using the Create method. If there are CREATEKEYS values associated with the Component Interface, these are the values you must set. If there are no CREATEKEYS values, you must set the required GETKEYS values instead. All keys required for creating a new Component Interface must be set before using the Create method, otherwise you receive a runtime error. If you do not use unique key values, (that is, you try to set the keys to values that already exist in the database) you receive a runtime error.

Setting the key values prior to using the Create method is analogous to filling in the key values in the Add dialog for a component when you access it in add mode.

Parameters

None.

Returns

A Boolean value: True if component was successfully created, False otherwise.

Example

```
&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.ACTION_REASON);
&MYCI.ACTION = "Additional Job";
&MYCI.ACTION_REASON = "0001";
&MYCI.Create();
```

Find

Syntax

Find()

Description

Note. Find can be used only at level 0 within the Component Interface.

You do not have to set values for all the key values. You can use the same wildcards in your Find that you can use in the search dialog, that is, % for one or more characters in a search pattern, and _ (underscore) for exactly one character. In addition, you can use partial values. For example, the following code finds all the data items where the employee ID starts with an "8":

```
&MYCI.Emplid = "8";
&MYDC = &MYCI.Find();
```

This is analogous to using a partial key from the search dialog, opening a component.

After you have a data collection, you can use one of the data collection methods to open the Component Interface.

Parameters

None.

Returns

A collection of Component Interfaces.

See Also

Chapter 10, Component Interface Classes , Data Collection Methods , page 805.

Get

Syntax

Get()

Description

The **Get** method associates the Component Interface object executing the method with an open Component Interface that matches the key values that were set prior to using the Get method. The key values you must set are the required GETKEYS values for the Component Interface.

Note. To retrieve all the history data for a Component Interface, you must specify the GetHistoryItems property as True *before* you use the **Get** method. If you want any PeopleCode programs associated with the fields to fire immediately after a value is changed, you must set the InteractiveMode property as True *before*you use the **Get** method.

After any execution of **Get**, you should check if there are any errors pending on the session object. In some special circumstances (involving failure of previously cached operations failing after the Get has executed) Get returns True even though the component wasn't retrieved.

Parameters

None.

Returns

A Boolean value: True if component was successfully retrieved, False otherwise.

Example

```
&MYCI.EMPLID = "8001";
&MYCI.Get();
If %Session.ErrorPending Then
   /* Get Unsuccessful, do error processing */
Else
   /* do regular processing */
End-if;
```

GetPropertyByName

Syntax

GetPropertyByName(*string*)

Description

The **GetPropertyByName** method returns the value of the property specified by *string*. For a collection, it returns a reference to the collection. Generally this function is used only in applications that cannot get the names of the component interface properties until runtime.

Parameters

Parameter	Description
string	The name of the property.

Returns

String. The value of the property.

ApiObject. For a collection. The value of the property.

Example

```
Local ApiObject &oSession, &oCI;
Local array of string &Keys_Arr, &Temp_Arr, &Prop_Arr;
Local string &strCIName, &PropertyValue;
Local number &I, &J, &K;

Function getPropertyValue(&oDataColl As ApiObject, &PropertyName As string)⇒
 Returns string
rem ***** Return property value
   Return &oDataColl.GetPropertyByName(&PropertyName);
End-Function;

Function getCollection1(&collectionName As string)
   Local ApiObject &oL1_DataColl, &oL1_DataItem;
rem ***** Return collection
   &oL1_DataColl = &oCI.GetPropertyByName(&collectionName);
   For &J = 1 To &oL1_DataColl.Count
      &oL1_DataItem = &oL1_DataColl.Item(&J);
      For &K = 1 To &Prop_Arr.Len
         &Temp_Arr = Split(&Prop_Arr [&K], "|");
         If &Temp_Arr [1] = "1" Then
            If &Temp_Arr [3] = "Property" Then
               &PropertyValue = getPropertyValue(&oL1_DataItem,
                   &Temp_Arr [2]);
            Else
               rem ***** Code to Get Collection 2 goes here *****;
            End-If;
         End-If;
      End-For;
   End-For;
End-Function;
```

See Also

Chapter 10, Component Interface Classes , SetPropertyByName, page 800.

Save

Syntax

Save()

Description

Saves any changes that have been made to the data of the Component Interface executing the method. You must save any new Component Interfaces you create before they are added to the database.

The standard PeopleSoft save business rules (that is, any PeopleCode programs associated with SaveEdit, SavePreChange, WorkFlow, and so on.) is executed after you execute this method. If you didn't specify the Component Interface to run in interactive mode, FieldEdit, FieldChange, and so on, also run at this time.

If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, you may want to delete it from the PSMessages collection.

Note. If you're running a Component Interface from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Parameters

None.

Returns

A Boolean value: True if component was successfully saved, False otherwise.

Example

```
&MYCI.Emplid = "8001";
&MYCI.Get();
&MYCI.CHECKLIST_CD = "00001";
&MYCI.Save;
```

See Also

Chapter 10, Component Interface Classes , InteractiveMode, page 804, Chapter 29, Session Class , ErrorPending, page 1567.

SetPropertyByName

Syntax

SetPropertyByName(*string, value***)**

Description

The **SetPropertyByName** method sets the value of the property specified by *string*. Generally this function is used only in applications that cannot set the names of the component interface properties until runtime.

Parameters

Parameter	Description
string	The name of the property.
value	The value to which the property is to be set.

Returns

None.

Example

```
Local ApiObject &oSession, &oCI;
Local array of string &Keys_Arr, &Temp_Arr, &Prop_Arr;
Local string &strCIName, &PropertyValue;
Local number &I, &J, &K;

&strCIName = "SDK_BUS_EXP";
&Keys_Arr = CreateArrayRept("", 0);
&Keys_Arr.Push("SDK_EMPLID|" | "8001");

&oSession = %Session;
&oCI = &oSession.GetCompIntfc(@("CompIntfc." | &strCIName));

For &I = 1 To &Keys_Arr.Len
   &Temp_Arr = Split(&Keys_Arr [1], "|");
   &oCI.SetPropertyByName(&Temp_Arr [1], &Temp_Arr [2]);
End-For;
```

See Also

Chapter 10, Component Interface Classes , GetPropertyByName, page 798.

Component Interface Class Properties

In this section, we discuss the Component Interface class properties. The properties are discussed in alphabetical order.

ComponentName

Description

This property returns the name of the Component Interface, as defined in PeopleSoft Application Designer, as a string.

This property is read-only.

CreateKeyInfoCollection

Description

This property returns a CompIntfPropInfoCollection collection that contains a CompIntfPropInfoCollection object for every key in CREATEKEYS.

This property is read-only.

Example

```
&CREATEKEYS = &CI.CreateKeyInfoCollection;
```

See Also

Chapter 10, Component Interface Classes , Accessing the Structure of a Component Interface , page 812.

EditHistoryItems

Description

This property works with the GetHistoryItems property to specify what data is accessed, and whether you can edit that data:

- If EditHistoryItems is False (the default) and GetHistoryItems is True, you access the data in the Component Interface in a similar manner as if you were accessing a component in **Update/Display All** mode. This means all history rows are returned, however, you can edit only rows with a date set in the future.
- If EditHistoryItems is True and GetHistoryItems is True, you access the data in the Component Interface in a similar manner as if you were accessing a component in **Correction** mode. This means all history rows are returned, and you can edit them.
- If GetHistoryItems is False, you access the data in the Component Interface in a similar manner as if you were accessing a component in **Update** mode. The EditHistoryItems has no effect when GetHistoryItems is False.

You must set this property to True *before* you execute the **Get** method.

This property is read-write.

See Also

Chapter 10, Component Interface Classes , GetHistoryItems, page 803.

FindKeyInfoCollection

Description

This property returns a CompIntfPropInfoCollection collection that contains a CompIntfPropInfoCollection object for every key in FINDKEYS.

This property is read-only.

See Also

Chapter 10, Component Interface Classes , Accessing the Structure of a Component Interface , page 812.

GetDummyRows

Description

When a new scroll is inserted on a page, that scroll is displayed even though it has no underlying data. Any scroll that is empty has one dummy row displayed with only the defaults set.

This property is True if the dummy row is to be displayed, False if you do not want to display the dummy row. The default value for this property is True.

This property is read-write.

Example

```
&MyCI.GetDummyRows = False;
```

GetHistoryItems

Description

This property works with the EditHistoryItems property to specify what data is accessed, and whether you can edit that data:

- If EditHistoryItems is False (the default) and GetHistoryItems is True, you access the data in the Component Interface in a similar manner as if you were accessing a component in **Update/Display All** mode. This means all history rows are returned, however, you can edit only rows with a date set in the future.
- If EditHistoryItems is True and GetHistoryItems is True, you access the data in the Component Interface in a similar manner as if you were accessing a component in **Correction** mode. This means all history rows are returned, and you can edit them.
- If GetHistoryItems is False, you access the data in the Component Interface in a similar manner as if you were accessing a component in **Update** mode. The EditHistoryItems has no effect when GetHistoryItems is False.

You must set this property to True *before* you execute the **Get** method.

This property is read-write.

Example

The following example checks the current status of the mode, then sets the GetHistoryItems and EditHistoryItems properties to True if the mode is Correction mode.

```
If %Mode = "C" Then
   &CI.EditHistoryItems = True;
   &CI.GetHistoryItems = True;
End-if;
```

See Also

Chapter 10, Component Interface Classes , EditHistoryItems, page 802.

GetKeyInfoCollection

Description

This property returns a CompIntfPropInfoCollection collection that contains a CompIntfPropInfoCollection object for every key in GETKEYS.

This property is read-only.

See Also

Chapter 10, Component Interface Classes , Accessing the Structure of a Component Interface , page 812.

InteractiveMode

Description

When this property is set as True, the Component Interface runs the same as a component: that is, any PeopleCode programs associated with FieldChange, RowInsert, and so on, run immediately after you make a change. If this property is set to False, these programs won't run until you execute the **Save** method.

This property is read-write.

See Also

Chapter 29, Session Class , ErrorPending, page 1567.

PropertyInfoCollection

Description

This property returns a CompIntfPropInfoCollection object for every property that isn't a collection (that is, a scroll.) If the property is a collection (scroll), use the **PropertyInfoCollection** property to get another collection.

This property is read-only.

See Also

Chapter 10, Component Interface Classes , Accessing the Structure of a Component Interface , page 812.

StopOnFirstError

Description

When this property is set as True, the CopyRowset (or CopyRowsetDelta) method currently executing halts its processing at the first error generated by the Component Interface.

When this property is set as True and InteractiveMode is set as False, processing of queued operations at save time is halted at the first error.

The default value is False.

This property is read-write.

Data Collection Methods

A *data collection* is the collection of data, available at runtime or during test mode, associated with a particular scroll (or record.) The data collection object returns information about every *row of data* (item) that is returned for that record at runtime.

To access a data collection, use the name of the record (scroll) as a property on a Component Interface.

See Also

Chapter 10, Component Interface Classes , Traversing a Component Interface and Using Data Collections , page 780.

CurrentItem

Syntax

```
CurrentItem()
```

Description

If the component associated with the Component Interface is effective-dated, **CurrentItem** returns a reference to the current effective-dated item (row of data). To get a specific item based on a date, use **GetEffectiveItem**.

If there is no current item, a Null is returned.

Example

```
&MYCD = &MYCI.EMPL_CHKLST_ITM;
&ITEM = &MYDC.CurrentItem();
```

DeleteItem

Syntax

```
DeleteItem(number)
```

Description

The **DeleteItem** method deletes the item (row of data) at the position specified by *number*. This parameter takes a number value.

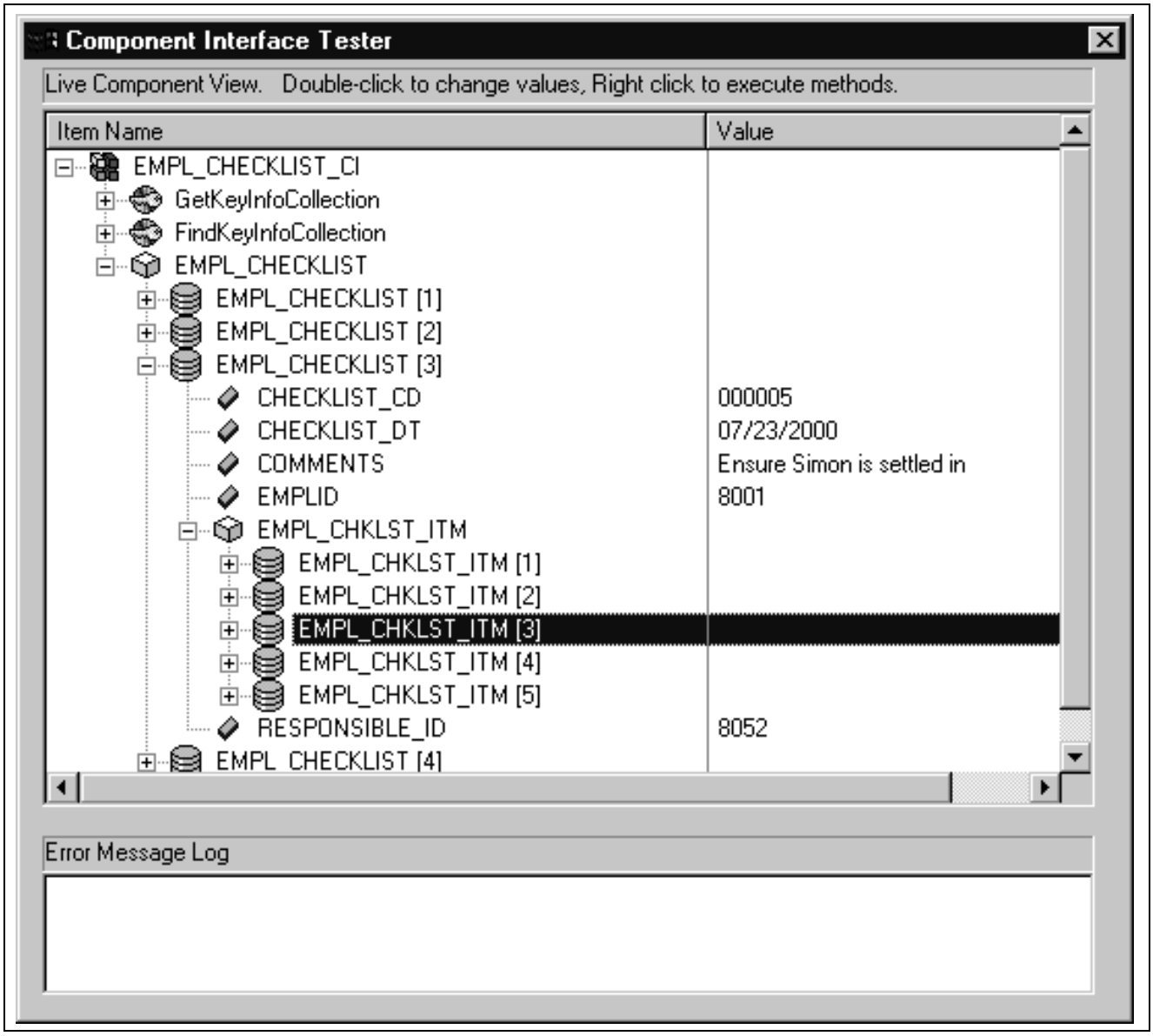
When the DeleteItem method is executed, if there are any RowDelete PeopleCode programs associated with any of the fields, they fire as well, as if the user pressed ALT+8 and ENTER or clicked the DeleteRow icon. However, the programs are executed as if turbo mode was selected. (In turbo mode default processing is performed *only* for the row being deleted.)

If you set the **InteractiveMode** property to True, any RowDelete PeopleCode runs immediately after you execute DeleteItem. If this property is set to False, any RowDelete PeopleCode runs after you execute the **Save** method.

The deleted item is not actually deleted from the database until after you use the **Save** method.

Example

For example, suppose your Component Interface had two scrolls: EMPL_CHECKLIST and EMPL_CHKLST_ITM. The data collection EMPL_CHECKLIST has four items (rows of data.) The data collection EMPL_CHKLST_ITM (under the third item) has five items (rows of data.) If you run this component in the Component Interface Tester, it would look as follows:



EMPL_CHK_BC in Component Interface Tester

To delete the third row in the third item, use the following:

```
Local ApiObject &MYSESSION;
Local ApiObject &MYCI;

&MYSESSION = %Session;
&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EMPL_CHK_BC);
&MYLVL1 = &MYCI.EMPL_CHECKLIST;
```

```
&ITEM2 = &MYLVL1.Item(3);
&MYLVL2 = &ITEM2.EMPL_CHKLST_ITM;
&MYLVL2.DeleteItem(3);
&MYCI.Save();
```

GetEffectiveItem

Syntax

GetEffectiveItem(*DateString, SeqNo*)

Description

If the component associated with the Component Interface is effective-dated, **GetEffectiveItem** returns a reference to the closest effective-dated item (row of data) that is less than or equal to the date specified by the *DateString*. To get an item based on the current effective date, use **CurrentItem**.

Note. *DateString* takes only a string value. You must convert a date variable into a string before you can use it for *DateString*. You can use the **String** function to do this.

Parameters

Parameter	Description
DateString	Specify the year, month, and day of the effective date that you want to look for. This parameter takes a string value. You can specify the date either as YYYY-MM-DD *or*MM/DD/YY.
SeqNo	Specify the sequence number of the effective date that you want to look for. This parameter takes a number value.

Returns

A reference to an effective-dated item.

Example

```
&MYCD = &MYCI.EMPL_CHKLST_ITM;
&DSTRING = String(&MYDATE);
&ITEM = &MYDC.GetEffectiveItem(&DSTRING, 1);
```

See Also

Chapter 1, PeopleCode Built-in Functions, String , page 451.

GetEffectiveItemNum

Syntax

GetEffectiveItemNum(*DateString, SeqNo*)

Description

If the component associated with the Component Interface is effective-dated, **GetEffectiveItemNum** returns a reference to the number of the closest effective-dated item (row of data) that is less than or equal to the date specified by the *DateString*. To get an item number based on the current effective date, use **CurrentItemNum**.

Note. *DateString* takes only a string value. You must convert a date variable into a string before you can use it for *DateString*. You can use the String function to do this.

Considerations for Returning Rows

GetEffectiveItemNum returns a valid row number only when EFFDT is less than or equal to *DateString*. If the value you use for *DateString* pre-dates all the rows in the data collection, this method returns a zero and logs a message in the PSMessages collection.

For example, if 12/01/1990 was the earliest date in the collection, the following would return zero:

```
&NUM = &MYDC.GetEffectiveItemNum("01/01/1900", 1);
```

Parameters

Parameter	Description
DateString	Specify the year, month, and day of the effective date that you want to look for. This parameter takes a string value. You can specify the date either as YYYY-MM-DD *or*MM/DD/YYYY.
SeqNo	Specify the sequence number of the effective date that you want to look for. This parameter takes a number value.

Returns

A number. This method returns 0 if no row matching the criteria is found.

Example

```
&MYCD = &MYCI.EMPL_CHKLST_ITM;
&DSTRING = String(&MYDATE);
&ITEMNUM = &MYDC.GetEffectiveItemNum(&DSTRING, 1);
```

See Also

Chapter 1, PeopleCode Built-in Functions, String , page 451.

InsertItem

Syntax

InsertItem(*number*)

Description

The **InsertItem** method inserts the item (row of data) at the position specified by *number*. This parameter takes a number value. You can nsert items below only the zero level scroll. If you need to add a new data item, use the **Create** method instead.

InsertItem adds the new row *after* the current row. If the row has an effective date (EFFDT) or an effective sequence (EFFSEQ), these values are copied into the new row.

If you specify −1 for *number*, InsertItem inserts a new item (row) after the *last* row.

The **InsertItem** method returns a reference to the newly created item (row of data).

When the InsertItem method is executed, if there are any RowInsert PeopleCode programs associated with any of the fields, they fire also, as if the user pressed ALT+7 and ENTER or clicked the InsertRow icon. However, the programs are executed as if turbo mode was selected. (In turbo mode default processing is performed *only* for the row being inserted.)

If you set the **InteractiveMode** property to True, any RowInsert PeopleCode runs immediately after you execute InsertItem. If this property is set to False, any RowInsert PeopleCode runs after you execute the **Save** method.

The inserted item is not added to the database until after you use the **Save** method.

Example

In the following example a new item (row of data) is added at the *end* of the current collection.

```
&MYDC = &MYCI.EMPL_CHECKLIST;
&COUNT = &MYDC.Count;
&ITEM = &MYDC.InsertItem(&COUNT);
&ITEM.CHECKLIST_CD = "00001";
&ITEM.RESPONSIBLE_ID = "6609";
&RSLT = &MYCI.Save();
```

Item

Syntax

Item(*number*)

Description

The **Item** returns the item (row of data) that exists at the *number* position in the data collection executing the method. The parameter takes a number value.

ItemByKeys

Syntax

ItemByKeys(*key_values*)

Description

The **ItemByKeys** method returns the item specified by the parameters. The number and type of keys are unique to each specific collection. Each key must be separated by a comma.

The collection reference must be the name of the Component Interface, followed by the record name. This method won't work on a collection reference (that is, &CI.RECNAME.ItemByKeys, not &MYCOLLECTION.ItemByKeys).

After you've returned an item, use the ItemNum property to determine the number of the item.

The keys must be in the *exact* order as in the Component Interface. A second level data collection also contains the keys of the parent data collection.

An easy way to determine the keys and their order in PeopleCode is to open the Component Interface in PeopleSoft Application Designer, and use the Test Component. To determine the keys in Visual Basic, use the Object Browser.

See Chapter 10, Component Interface Classes , ItemNum, page 812.

To see the signature for ItemByKeys:

1. Open the Component Interface in PeopleSoft Application Designer.
2. Start the Component Interface Tester.

   Select the open Component Interface, then right-click, and select Test Component Interface from the pop-up menu.
3. Instantiate an object.

   Add data to the Get or Create keys and click **Get Existing** or **Create New**, respectively.
4. Expand the instantiated component until you find the collection in which you're interested.
5. Right-click on the collection and select ItemByKeys from the resulting pop-up menu.
6. The dialog that follows shows you the specific parameters, types, and order in which you should call ItemByKeys.

Returns

An item (row) of data from a data collection.

Example

```
Local ApiObject &MYSESSION;
Local ApiObject &CI;
Local ApiObject &CI_COLLECTION;
Local ApiObject &CI_ITEM;

&MYSESSION = %Session;
&CI = &MYSESSION.GetCompIntfc(COMPINTFC.CM_EVAL);
&CI.EMPLID = "8001";
If &CI.Get() <> 1 Then
   Exit;
End-If;

&CI_COLLECTION = &CI.CM_EVALUATIONS;
&COUNT = &CI_COLLECTION.Count;

&CI_ITEM = &CI.CM_EVALUATIONS.itembykeys("01");
&CI_ITEM.DESCR50 = "TEST";
If &CI.Save() <> 1 Then
   Exit;
End-If;
```

See Also

PeopleSoft Component Interfaces, Developing the Component Interface, Testing the Component Interface.

Data Collection Properties

This section explains the following Data Collection properties:

- Count
- CurrentItemNumber

Count

Description

This property returns the number of data items (rows of data) in the data collection.

This property is read-only.

Example

```
&CI = &MYSESSION.GetCompIntfc(COMPINTFC.CM_EVAL_BC);
&CI.EMPLID = "8001";
&CI.Get()

&CI_COLLECTION = &CI.CM_EVALUATIONS;
&COUNT = &CI_COLLECTION.Count;
```

CurrentItemNumber

Description

If the component associated with the Component Interface is effective-dated, this property returns the item number for the current effective-dated item (row of data).

This property is read-only.

Data Item Class Property

This section discusses the ItemNum property.

ItemNum

Description

This property returns the number of the data item (row) in the collection. For example, many of the data collection methods takes a number as a parameter. Use this property to determine the item number (row number) of an item in a collection, then use that number in another method.

This property is read-only.

Example

```
Evaluate USER_ACTION

.  .  .

When = "D"
   &CI_ITEM = &CI_LVL1_NAMES.ItemByKeys(&NAME_TYPE, &NAME_PART);
   If &CI_ITEM <> Null then
      &I = &CI_LVL1_NAMES.ItemNum;
      &CI_LVL1_NAMES.DeleteItem(&I);
   End-if;
.  .  .
End-Evaluate;
```

Accessing the Structure of a Component Interface

The structure of a Component Interface can be accessed using a CompIntfPropInfoCollection object. You access a CompIntfPropInfoCollection object from a CompIntfPropInfoCollection collection. There is more than one way to instantiate a CompIntfPropInfoCollection collection.

Note. You don't have to populate a Component Interface with data before you access the structure. You can access the structure of a Component Interface immediately after you use the GetCompIntfc method with a session object. Accessing the structure of a Component Interface before you populate it with data may increase your performance.

See Chapter 10, Component Interface Classes , GetCompIntfc, page 785.

CompIntfPropInfoCollection Collection

There are two types of CompIntfPropInfoCollection object properties: *field* properties and *collection* properties.

A *field* property maps to a specific record field. You can access structural information about the field using a CompIntfPropInfoCollection object. This information includes the name of the field, whether it's required, is it based on a prompt table, and so on.

A *collection* property is just that, a collection of properties. And before you can access a CompIntfPropInfoCollection object, you must first get a CompIntfPropInfoCollection *collection.* The following are the valid types of CompIntfPropInfoCollection collections:

- CREATEKEYS, GETKEYS and FINDKEYS

  When you create a component, you must specify the search record to be used with that component. You can also specify an alternate search record to be used when the component is accessed in Add mode. The key fields from those records make up GETKEYS, FINDKEYS, and CREATEKEYS collections.

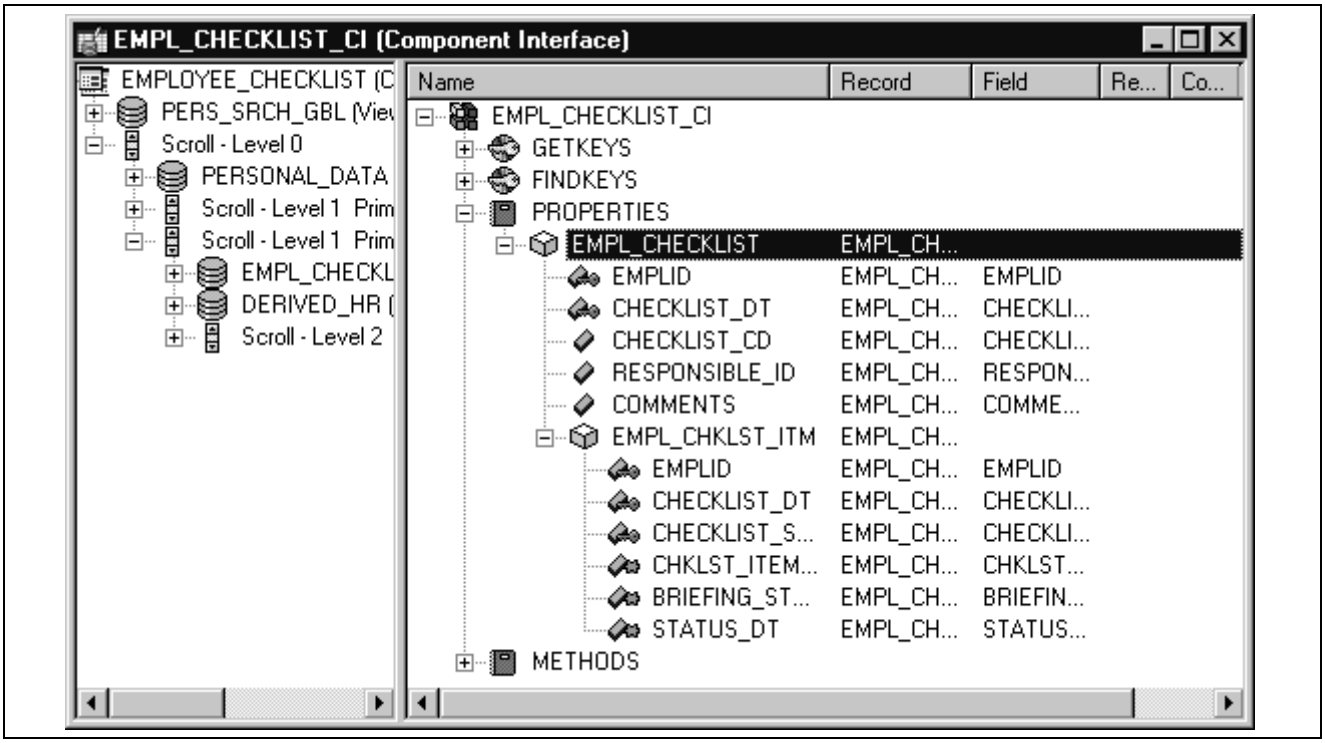
  - CREATEKEYS: A collection containing the key fields of the search record specified to be used in Add mode. Use the **CreateKeyInfoCollection** property to instantiate the CompIntfPropInfoCollection collection.
  - GETKEYS: A collection containing the primary required key fields from the primary search record. Use the **GetKeyInfoCollection** property to instantiate the CompIntfPropertInfo collection.
  - FINDKEYS: A collection containing the key fields and the alternate key fields from the primary search record. Use the **FindKeyInfoCollection** property to instantiate the CompIntfPropertInfo collection.

- A page scroll

  The fields associated with a page scroll are in this type of collection. This may or may not be all the fields associated with the record. Use the **PropertyInfoCollection** property to instantiate this kind of collection.

If the page the Component Interface is based on contains a secondary scroll, you can check the **Type** property to determine if the object is a CompIntfPropInfoCollection object (that is, a field), or a scroll. Then, to get the properties of the fields associated with that secondary scroll, you can use the **PropertyInfoCollection** property on the CompIntfPropInfoCollection object.

For example, the following Component Interface has a level 0 and level 1 scroll.



Component Interface with secondary scroll

The level 0 scroll is made up of three fields: CHECKLIST_CD, RESPONSIBLE_ID, COMMENTS, and a level 1 scroll, EMPL_CHKLST_ITM. The CompIntfPropInfoCollection collection for this Component Interface would have four items in it. Three would be CompIntfPropInfoCollection objects (the three fields.) The last item, EMPL_CHKLST_ITM would *not* be a valid CompIntfPropInfoCollection object. You can use the **IsCollection** property to verify if an item in a collection is itself a collection or a valid CompIntfPropInfoCollection object.

To access the fields in a lower level scroll, you must use the **PropertyInfoCollection** property first, to return a collection of those fields.

The following example loops through a Component Interface. It pulls out the names of the properties in the first three levels of a Component Interface. If the property is a nested collection, it prefixes the ancestor collection name to the property name.

```
&MYSESSION = %Session;
&CI = &MYSESSION.GetCompIntfc(COMPINTFC.VOLNEW);
&PROPINFO_0 = &CI.PropertyInfoCollection;
For &I = 1 To &PROPINFO_0.Count;
   &PROPITEM_0 = &PROPINFO_0.Item(&I);
   Warning (&PROPITEM_0.Name);
   If (&PROPITEM_0.IsCollection) Then
      &PROPINFO_1 = &PROPITEM_0.PropertyInfoCollection;
      For &J = 1 To &PROPINFO_1.Count;
         &PROPITEM_1 = &PROPINFO_1.Item(&J);
         &S1 = &PROPITEM_0.Name | "." | &PROPITEM_1.Name;
         Warning (&S1);
         If (&PROPITEM_1.IsCollection) Then
            &PROPINFO_2 = &PROPITEM_1.PropertyInfoCollection;
            For &K = 1 To &PROPINFO_2.Count;
               &PROPITEM_2 = &PROPINFO_2.Item(&K);
               &S1 = &PROPITEM_0.Name | "." | &PROPITEM_1.Name | "." | ⇒
&PROPITEM_2.Name;
               Warning (&S1);
            End-For;
         End-If
      End-For;
   End-If;
End-For;
```

CompIntfPropInfoCollection Collection Methods

This section explains the CompIntfPropInfoCollection collection methods.

First

Syntax

First()

Description

The **First** method returns the first CompIntfPropInfoCollection object in the CompIntfPropInfoCollection collection object executing the method.

Example

```
&CIINFO = &CIPROPCOLL.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns a CompIntfPropInfoCollection object that exists at the *number* position in the CompIntfPropInfoCollection collection executing the method

If the specified item is *itself* a collection, the CompIntfPropInfoCollection object that gets returned is invalid (set to NULL). You can use the **PropertyInfoCollection** property to return a collection of CompIntfPropInfoCollection objects for the collection.

Example

```
&SCROLL1 = &MYCI.PropertyInfoCollection;
For &I = 1 to &SCROLL1.Count;
&PROPERTY = &SCROLL1.Item(&I);
If &PROPERTY.IsCollection Then
&SCROLL2 = &PROPERTY.PropertyInfoCollection;
/*do scroll 2 processing */
Else
/* do scroll 1 processing */
End-If;
End-For;
```

Next

Syntax

Next()

Description

The **Next** method returns the next CompIntfPropInfoCollection object in the CompIntfPropInfoCollection collection object executing the method. You can use this method only after you have used either the **First** or **Item** methods: otherwise the system doesn't know where to start.

CompIntfPropInfoCollection Collection Properties

This section explains the Count property.

Count

Description

This property returns the number of CompIntfPropInfoCollection objects in the CompIntfPropInfoCollection collection object executing the method.

This property is read-only.

Example

```
&COUNT = &MYCIPROPINFOC.Count;
```

CompIntfPropInfoCollection Object Properties

This section explains the CompIntfPropInfoCollection Object properties.

Format

Description

This property returns the field format for the object executing the property (that is, name, phone, zip, SSN, and so on) as a number. The values are:

Value	Description
0	No Format
1	Name
2	Phone
3	Zip
4	SSN
5	Routine
6	Mixed Case
7	Raw Binary
8	Number only
9	SIN
10	Phone International
11	Zip International
12	Seconds
13	Microseconds
14	Custom

This property is read-only.

IsCollection

Description

This property returns True if the object executing the property is a data collection, False otherwise. If IsCollection is True, other field-oriented properties like Required, Type, Xlat, YesNo, Prompt and Format are undefined. If IsCollection is False, the object represents a field and all the previous properties are defined as described.

This property is read-only.

IsReadOnly

Description

This property returns True if the property marked Read Only in the Component Interface Definition; False otherwise.

This property is read-only.

Key

Description

This property returns True if the object executing the property is a key, False otherwise.

This property is read-only.

LabelLong

Description

This property returns the record field LongName value as a string. If there is a component override for this value, it is *not* included.

This property is read-only.

LabelShort

Description

This property returns the record field ShortName value as a string. If there is a component override for this value, it is *not* included.

This property is read-only.

Length

Description

This property returns the length of the field property as a number. If the property is a collection, this property returns a zero.

The length of the field is calculated by converting it to a string then getting the length of the string. This means that *everything* in the field is counted as a character, including spaces, date time separation characters, decimal points, sign indicators, and so on.

For example, the following string has 10 characters:

```
01/01/2001
```

The following string has five characters:

```
10.10
```

The following string has six characters:

```
-10.10
```

This property is read-only.

Name

Description

This property returns the name of the object executing the property as a string.

This property is read-only.

Prompt

Description

This property returns True if the object executing the property is associated with a prompt table, False otherwise.

This property is read-only.

PropertyInfoCollection

Description

This property returns another CompIntfPropInfoCollection collection if the object executing the property is a collection.

This property is read-only.

Example

```
&SCROLL1 = &MYCI.PropertyInfoCollection;
For &I = 1 to &SCROLL1.Count;
   &PROPERTY = &SCROLL1.Item(&I);
```

```
   If &PROPERTY.IsCollection Then
   &SCROLL2 = &PROPERTY.PropertyInfoCollection;
   /*do scroll 2 processing */
   Else
/* do scroll 1 processing */
   End-If;
End-For;
```

Required

Description

This property returns True if the object executing the property is a required property, False otherwise.

This property is read-only.

Type

Description

This property returns the field type, as a number, of the object. The values are:

Value	Description
0	Character
1	Long Character
2	Number
3	Signed Number
4	Date
5	Time
6	DateTime
7	SubRecord (Not supported with Component Interfaces)
8	Image (Limited support with Component Interfaces)
9	ImageReference (Not supported with Component Interfaces)

This property is read-only.

Xlat

Description

This property returns True if the object executing the property is associated with an XLAT table, False otherwise.

This property is read-only.

YesNo

Description

This property returns True if the object executing the property is associated with the Yes/No table, False otherwise.

This property is read-only.

Component Interface Examples

The following are examples of some of the most usual actions you're likely to perform using a Component Interface.

Create a New Instance of Data Example

The following is an example of how to create a new instance of a Component Interface.

To create a new instance of data:

In this example, you are creating a new instance of data for the EXPRESS Component Interface, which is based on the EXPRESS_ISSUE_INV component. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MYSESSION;
Local ApiObject &MYCI;


&MYSESSION = %Session;
&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EXPRESS);
&MYCI.BUSINESS_UNIT = "H01B";
&MYCI.INTERNAL_FLG = "Y";
&MYCI.ORDER_NO = "NEXT';
&MYCI.Create();
&MYCI.CUSTOMER = "John's Chicken Shack";
&MYCI.LOCATION = "H10B6987";
.
.
.
If NOT(&MYCI.Save()) Then
   /* save didn't complete */
   &COLL = &MYSESSION.PSMessages;
   For &I = 1 to &COLL.Count
      &ERROR = &COLL.Item(&I);
      &TEXT = &ERROR.Text;
      /* do error processing */
   End-For;
   &COLL.DeleteAll();
End-if;
```

1. Get a session object.

   Before you can get a Component Interface, you have to get a session object. The session controls access to the Component Interface, provides error tracing, enables you to set the runtime environment, and so on.

   ```
   &MYSESSION = %Session;
   ```

2. Get a Component Interface.

   Use the GetCompIntfc method with a session object to get the Component Interface. You must specify a Component Interface definition that has already been created. You receive a runtime error if you specify a Component Interface that doesn't exist.

   ```
   &MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EXPRESS);
   ```

   After you execute the GetCompIntfc method, you have only the *structure* of the Component Interface. You haven't populated the Component Interface with data yet.

3. Set the CREATEKEYS.

   These key values are required to open a new instance of the data. If the values you specify aren't unique, that is, if an instance of the data already exists in the database with those key values, you receive a runtime error.

   ```
   &MYCI.BUSINESS_UNIT = "H01B";
   &MYCI.INTERNAL_FLG = "Y";
   &MYCI.ORDER_NO = "NEXT';
   ```

4. Create the instance of data for the Component Interface.

   After you set the key values, you must use the Create method to populate the Component Interface with the key values you set.

   ```
   &MYCI.Create();
   ```

   This creates an instance of this data. However, it hasn't been saved to the database. You must use the Save method before the instance of data is committed to the database.

5. Set the rest of the fields.

   Assign values to the other fields.

   ```
   &MYCI.CUSTOMER = "John's Chicken Shack";
   &MYCI.LOCATION = "H10B6987";
   .
   .
   .
   ```

   If you have specified InteractiveMode as True, every time you assign a value or use the InsertItem or DeleteItem methods, any PeopleCode programs associated with that field (either with record field or the component record field) fires (FieldEdit, FieldChange, RowInsert, and so on.)

6. Save the data.

   When you execute the Save method, the new instance of the data is saved to the database.

   ```
   If NOT(&MYCI.Save()) Then
   ```

   The **Save** method returns a Boolean value: True if the save was successful, False otherwise. You can use this value to do error checking.

The standard PeopleSoft save business rules (that is, any PeopleCode programs associated with SaveEdit, SavePreChange, WorkFlow, and so on.) are executed. If you didn't specify the Component Interface to run in interactive mode, FieldEdit, FieldChange, and so on, also run at this time for all fields that had values set.

Note. If you're running a Component Interface from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

---

7. Check Errors.

   You can check if there were any errors using the PSMessages property on the session object.

```
If NOT(&MYCI.Save()) Then
   /* save didn't complete */
   &COLL = &MYSESSION.PSMessages;
   For &I = 1 to &COLL.Count
      &ERROR = &COLL.Item(&I);
      &TEXT = &ERROR.Text;
      /* do error processing */
   End-For;
   &COLL.DeleteAll();
End-if;
```

   If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, delete it from the PSMessages collection.

   The **Text** property of the PSMessage returns the text of the error message. At the end of this text is a contextual string that contains the name of the field that generated the error. The contextual string has the following syntax:

```
{BusinessComponentName.[CollectionName(Row).[CollectionName(Row).⇒
[CollectionName(Row)]]].PropertyName}
```

   For example, if you specified the incorrect format for a date field of the Component Interface named ABS_HIST, the **Text** property would contain the following string:

```
Invalid Date {ABS_HIST.BEGIN_DT} (90), (1)
```

   The contextual string (by itself) is available using the **Source** property of the PSMessage.

Note. If you've called your Component Interface from an Application Engine program, all errors are also logged in the Application Engine error log tables.

See Chapter 29, Session Class , Error Handling , page 1560 and Chapter 29, Session Class , Source, page 1574.

Getting an Existing Instance of Data Example

The following is an example of how to get an existing instance of a Component Interface.

To get an existing instance of data:

In this example, you are getting an existing instance of data for the EMPL_CHKLST_BC Component Interface, which is based on the EMPLOYEE_CHECKLIST component. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MYSESSION;
Local ApiObject &MYCI;

&MYSESSION = %Session;
&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EMPL_CHKLST_BC);
&MYCI.EMPLID= "8001";
&MYCI.Get();

/* Get checklist Code */

&CHECKLIST_CD = &MYCI.CHECKLIST_CD;

/* Set Effective date */

&MYCI.EFFDT = "05-01-1990";
.
.
.
If NOT(&MYCI.Save()) Then
   /* save didn't complete */
   &COLL = &MYSESSION.PSMessages;
   For &I = 1 to &COLL.Count
      &ERROR = &COLL.Item(&I);
      &TEXT = &ERROR.Text;
      /* do error processing */
   End-For;
   &COLL.DeleteAll();
End-if;
```

1. Get a session object.

   Before you can get a Component Interface, you have to get a session object. The session controls access to the Component Interface, provides error tracing, enables you to set the runtime environment, and so on.

   ```
   &MYSESSION = %Session;
   ```

2. Get a Component Interface.

   Use the GetCompIntfc method with a session object to get the Component Interface. You must specify a Component Interface definition that has already been created. You receive a runtime error if you specify a Component Interface that doesn't exist.

   ```
   &MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EMPL_CHKLST_BC);
   ```

   After you execute the GetCompIntfc method, you have only the *structure* of the Component Interface. You haven't populated the Component Interface with data yet.

3. Set the GETKEYS.

   These are the key values required to return a unique instance of existing data. If the keys you specify allow for more than one instance of the data to be returned, or if no instance of the data matching the key values is found, you receive a runtime error.

   ```
   &MYCI.EMPLID = "8001";
   ```

4. Get the instance of data for the Component Interface.

   After you set the key values, you have to use the Get method.

   ```
   &MYCI.Get();
   ```

   This populates the Component Interface with data, based on the key values you set.

5. Get field values or set field values.

   At this point, you can either get values or set values.

   ```
   &CHECKLIST_CD = &MYCI.CHECKLIST_CD;

   /* OR */
   &MYCI.EFFDT = "05-01-1990";
   ```

   If you have specified InteractiveMode as True, every time you assign a value, any PeopleCode programs associated with that field (either with record field or the component record field) fires (FieldEdit, FieldChange, and so on.)

6. Save or Cancel the Component Interface, as appropriate.

   If you've changed values and you want to save your changes to the database, you must use the **Save** method.

   ```
   If NOT(&MYCI.Save()) Then
   ```

   The **Save** method returns a Boolean value: True if the save was successful, False otherwise. Use this value to do error checking.

   The standard PeopleSoft save business rules (that is, any PeopleCode programs associated with SaveEdit, SavePreChange, WorkFlow, and so on.) are executed. If you didn't specify the Component Interface to run in interactive mode, FieldEdit, FieldChange, and so on, also run at this time.

   Note. If you're running a Component Interface from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

   If you don't want to save any changes to the data, use the **Cancel** method. The Component Interface ise reset to the state it was in just after you used the GetCompIntfc method.

   ```
   &MYCI.Cancel();
   ```

   This is similar to a user pressing ESC while in a component, and choosing to not save any of their changes.

7. Check Errors.

   You can check if there were any errors using the PSMessages property on the session object.

   ```
   If NOT(&MYCI.Save()) Then
      /* save didn't complete */
      &COLL = &MYSESSION.PSMessages;
      For &I = 1 to &COLL.Count
         &ERROR = &COLL.Item(&I);
         &TEXT = &ERROR.Text;
         /* do error processing */
      End-For;
      &COLL.DeleteAll();
   ```

```
End-if;
```

If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, delete it from the PSMessages collection.

The **Text** property of the PSMessage returns the text of the error message. At the end of this text is a contextual string that contains the name of the field that generated the error. The contextual string has the following syntax:

```
{ComponentInterfaceName.[CollectionName(Row).[CollectionName(Row).⇒
[CollectionName(Row)]]].PropertyName}
```

For example, if you specified the incorrect format for a date field of the Component Interface named ABS_HIST, the **Text** property would contain the following string:

```
Invalid Date {ABS_HIST.BEGIN_DT} (90), (1)
```

The contextual string (by itself) is available using the **Source** property of the PSMessage.

Note. If you've called your Component Interface from an Application Engine program, all errors are also logged in the Application Engine error log tables.

See Chapter 29, Session Class , Error Handling , page 1560 and Chapter 29, Session Class , Source, page 1574.

Retrieving a List of Instance of Data Example

The following is an example of how to retrieve a list of instances of data.

To retrieve a list of instances of data:

In this example, you are getting a list of existing instances of data for the EMPL_CHKLST_CI Component Interface, which is based on the EMPLOYEE_CHECKLIST component. The following is the complete code sample: the steps break it up and explain each line.

```
Local ApiObject &MYSESSION;
Local ApiObject &MYCI;
Local ApiObject &MYNEWCI;


&MYSESSION = %Session;
&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EMPL_CHKLST_CI);
&MYCI.EMPLID= "8";
&MYCI.LAST_NAME_SRCH = "S";
&MYLIST = &MYCI.Find();
For &I = 1 to &MYLIST.Count;

   /* note: do not reuse the CI used to create the list, or the list will be⇒
 destroyed */

&MYNEWCI = &MYLIST.Item(&I);

/* CI from list still must be instantiated to use it */
```

```
&MYNEWCI.Get();

/* do some processing */

End-For;
```

1. Get a session object.

   Before you can get a Component Interface, you have to get a session object. The session controls access to the Component Interface, provides error tracing, enables you to set the runtime environment, and so on.

   ```
   &MYSESSION = %Session;
   ```

2. Get a Component Interface.

   Use the GetCompIntfc method with a session object to get the Component Interface. You must specify a Component Interface definition that has already been created. You receive a runtime error if you specify a Component Interface that doesn't exist.

   ```
   &MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.EMPL_CHKLST_CI);
   ```

   After you execute the GetCompIntfc method, you have only the *structure* of the Component Interface. You haven't populated the Component Interface with data yet.

3. Set the FINDKEYS values.

   These can be alternate key values or partial key values. If no instance of the data matching the key values is found, you receive a runtime error.

   ```
   &MYCI.EMPLID = "8";
   &MYCI.LAST_NAME_SRCH = "S";
   ```

4. Get a list of data instances for the Component Interface.

   After you set the alternate or partial key values, use the Find method to return a list of data instances for the Component Interface.

   ```
   &MYLIST = &MYCI.Find();
   ```

   Note. The Find method can be executed only at level 0 in a Component Interface.

5. Select an instance of the data.

   To select an instance of the data, you can use any of the following standard data collection methods:

- First()
- Item(*number*)
- Next()

For example, you could loop through every instance of the data to do some processing:

```
For &I = 1 to &MYLIST.Count;
   /* note: do not reuse the Component Interface used to */
   /* create the list here, or the list will be destroyed */
&MYNEWCI = &MYLIST.Item(&I);
```

```
/* CI from list must still be instantiated to use it */
&MYNEWCI.Get();
/* do some processing */
End-For;
```

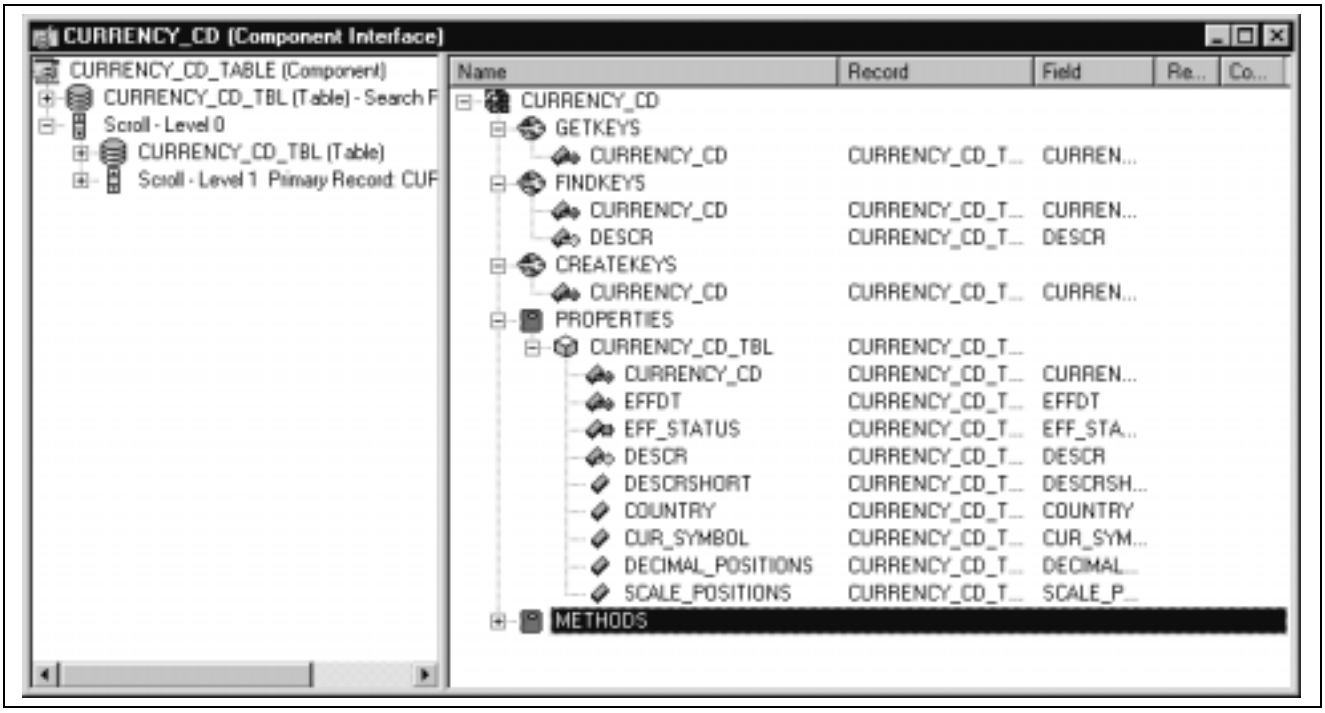
After you have a specific instance of the data, you can get values, set values, and so on.

See Chapter 10, Component Interface Classes , Data Collection Methods , page 805.

Inserting Effective-Dated Data Example

You can rename a collection in a Component Interface. For example, suppose you had the same record at level 0 and at level 1. You may want to rename the level 1 data collection to reflect this. The data in a data collection is associated with the primary database record of a scroll, *not* with the name you supply.

Here is an example of using a Component Interface that has the same record at level 0 and level 1. The Component Interface is based on the CURRENCY_CD_TBL component.



Example of CI with same record at level 0 and level 1

The following code example is based on this Component Interface and does the following:

1. Gets an existing Component Interface.
2. Finds the current effective-dated item index.
3. Inserts a new row before the current effective-dated item using the InsertItem method.

```
Local ApiObject &Session;
Local ApiObject &CURRENCY_CD;
Local ApiObject &CURRENCY_CD_TBLCol;
Local ApiObject &CURRENCY_CD_TBLItm;
Local ApiObject &PSMessages;
```

```
Local string &ErrorText, &ErrorType;
Local number &ErrorCount;
Local Boolean &Error;

Function CheckErrorCodes()

   &PSMessages = &Session.PSMessages;
   &ErrorCount = &PSMessages.Count;
   For &i = 1 To &ErrorCount
      &ErrorText = &PSMessages.Item(&i).Text;
      &ErrorType = &PSMessages.Item(&i).Type;
   End-For;

End-Function;

/* Initialize Flags */
&Error = False;

&Session = %Session;

If &Session <> Null Then

   CheckErrorCodes();
   /* Application Specific Error Processing */

Else

   &CURRENCY_CD = &Session.GetCompIntfc(CompIntfc.CURRENCY_CD);

   If &CURRENCY_CD = Null Then

      CheckErrorCodes();
      /* Application Specific Error Processing */

   Else

      /* Set Component Interface Get Keys */
      &CURRENCY_CD.CURRENCY_CD = "USD";

      If Not &CURRENCY_CD.Get() Then

         CheckErrorCodes();
         /* Application Specific Error Processing */
         &Error = True;

      End-If;

      If Not &Error Then

         &CURRENCY_CD_TBLCol = &CURRENCY_CD.CURRENCY_CD_TBL;
```

```
         &CURRENCY_CD_TBLItm = &CURRENCY_CD_TBLCol.InsertItem⇒
(&CURRENCY_CD_TBLCol.CurrentItemNum());
         &CURRENCY_CD_TBLItm.EFFDT = %Date;
         &CURRENCY_CD_TBLItm.EFF_STATUS = "A";
         &CURRENCY_CD_TBLItm.DESCR = "NewCurrencyCode";
         &CURRENCY_CD_TBLItm.DESCRSHORT = "New";
         &CURRENCY_CD_TBLItm.COUNTRY = "USA";
         &CURRENCY_CD_TBLItm.CUR_SYMBOL = "?";
         &CURRENCY_CD_TBLItm.DECIMAL_POSITIONS = 4;
         &CURRENCY_CD_TBLItm.SCALE_POSITIONS = 0;

         /* Save Instance of Component Interface */
         If Not &CURRENCY_CD.Save() Then

            CheckErrorCodes();
            /* Application Specific Error Processing */

         End-If;

      End-If;
      /* End: Set Component Interface Properties */

      /* Cancel Instance of Component Interface */
      &CURRENCY_CD.Cancel();

   End-If;

End-If;
```

Inserting Effective-Dated Data Example Using Visual Basic

Here's a code example in Visual Basic that does the same thing as the previous code example.

```
Private Sub CURRENCY_CD()

        On Error GoTo eMessage

        '***** Set Object References *****
        Dim oCISession As Object
        Dim oCURRENCY_CD As Object
        Dim oCURRENCY_CD_TBL As Object
        Dim oCURRENCY_CD_TBLItem As Object

        '***** Set Connect Parameters *****
        strAppSeverPath = "//PSOFT101999:9000"
        strOperatorID = "PTDMO"
        strPassword = "PTDMO"

        '***** Create PeopleSoft Session Object *****
        Set oCISession = CreateObject("PeopleSoft.Session")
```

```
        '***** Connect to the App Sever *****
        oCISession.Connect 1, strAppSeverPath, strOperatorID, strPassword, 0

        '***** Get the Component *****
        Set oCURRENCY_CD = oCISession.GetCompIntfc("CURRENCY_CD")

        '***** Set the Component Interface Mode *****
        oCURRENCY_CD.InteractiveMode = False
        oCURRENCY_CD.GetHistoryItems = True

        '***** Set Component Get/Create Keys *****
        oCURRENCY_CD.CURRENCY_CD = "USD"

        '***** Execute Create *****
        oCURRENCY_CD.Get

        'Set CURRENCY_CD_TBL Collection Field Properties --
    'Parent: PS_ROOT Collection
        Set oCURRENCY_CD_TBL = oCURRENCY_CD.CURRENCY_CD_TBL
        Set oCURRENCY_CD_TBLItem = oCURRENCY_CD_TBL.InsertItem(oCURRENCY_CD_⇒
TBL.CurrentItemNum())

        oCURRENCY_CD_TBLItem.EFFDT = Date
        oCURRENCY_CD_TBLItem.EFF_STATUS = "A"
        oCURRENCY_CD_TBLItem.DESCR = "NewCurrencyCode"
        oCURRENCY_CD_TBLItem.DESCRSHORT = "New"
        oCURRENCY_CD_TBLItem.COUNTRY = "USA"
        oCURRENCY_CD_TBLItem.CUR_SYMBOL = "?"
        oCURRENCY_CD_TBLItem.DECIMAL_POSITIONS = 4
        oCURRENCY_CD_TBLItem.SCALE_POSITIONS = 0

        '***** Save Component Interface *****
        oCURRENCY_CD.Save
        oCURRENCY_CD.Cancel

        Exit Sub

eMessage:
        '***** Display VB Runtime Errors *****
        MsgBox Err.Description

        '***** Display PeopleSoft Error Messages *****
        If oCISession.PSMessages.Count > 0 Then
                For i = 1 To oCISession.PSMessages.Count
                        MsgBox oCISession.PSMessages.Item(i).Text
                Next i
        End If

End Sub
```

```
Sub MAIN()
    CURRENCY_CD
End Sub
```

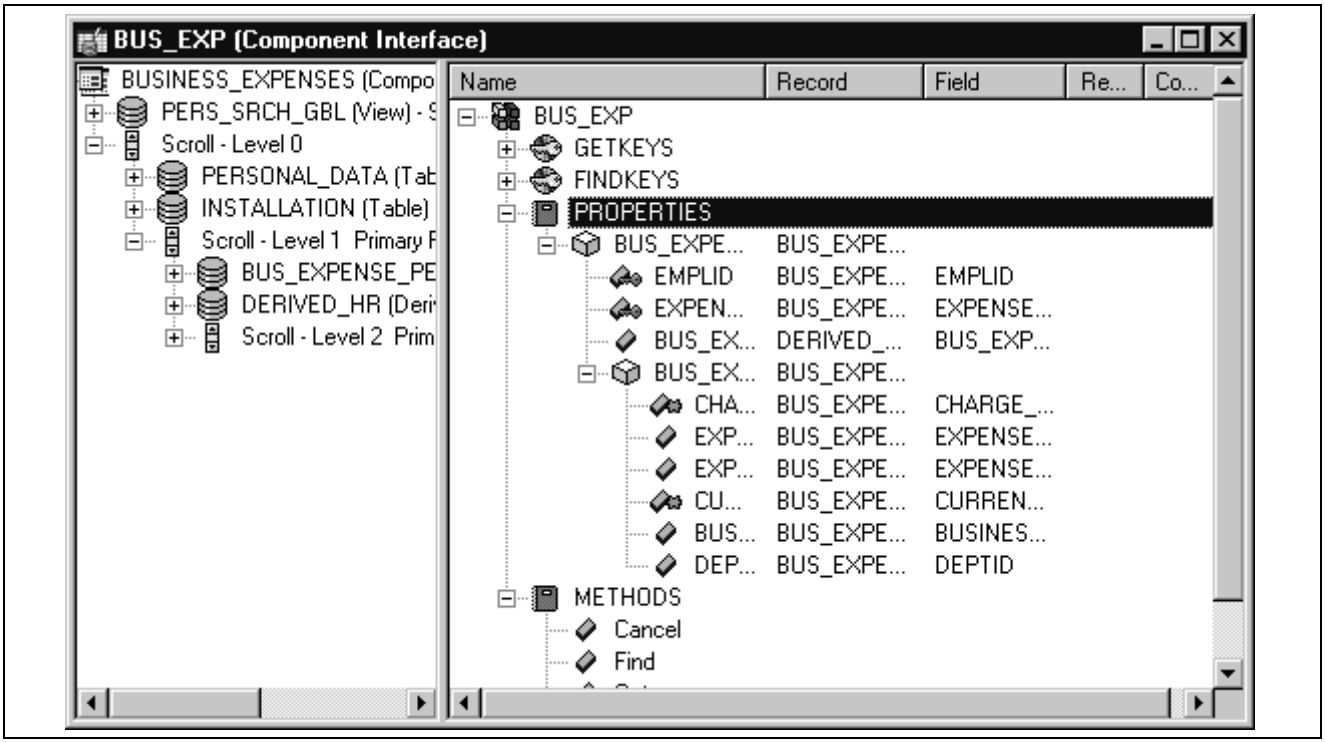
Inserting or Deleting a Row of Data Example

The CopyRowset and CopyRowsetDelta methods use the primary database name of a scroll, not the name you may give a collection.

A data collection represents a row of data. You often insert or delete a row of data.

To insert or delete a row of data:

In this example, you are getting a existing instance of data for the BUS_EXP Component Interface, which is based on the BUSINESS_EXPENSES component, then inserting (or deleting) a row of data in the second level scroll.



BUS_EXP Component Interface definition

The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MYSESSION;
Local ApiObject &MYCI;

&MYSESSION = %Session;
&MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.BUS_EXP);
&MYCI.EMPLID= "8001";
&MYCI.Get();
&MYLEVEL1 = &MYCI.BUS_EXPENSE_PER;
/* get appropriate item in lvl1 collection */
```

```
For &I = 1 to &MYLEVEL1.Count
&ITEM = &MYLEVEL1.Item(&I);
&MYLEVEL2 = &ITEM.BUS_EXPENSE_DTL;
&COUNT = &MYLEVEL2.Count
/* get appropriate item in lvl2 collection */
For &J = 1 to &COUNT
&LVL2ITEM = &MYLEVEL2.Item(&J);
&CIDATE = &LVL2ITEM.CHARGE_DT;
If &CIDATE <> %Date Then
      /* insert row */
&NEWITEM = &MYLEVEL2.InsertItem(&COUNT);
   /* set values for &NEWITEM */
Else
      /* delete last row */
   &MYLEVEL2.DeleteItem(&COUNT);
End-If;
   End-For;
End-For;

If NOT(&MYCI.Save()) Then
   /* save didn't complete */
   &COLL = &MYSESSION.PSMessages;
   For &I = 1 to &COLL.Count
      &ERROR = &COLL.Item(&I);
      &TEXT = &ERROR.Text;
      /* do error processing */
   End-For;
   &COLL.DeleteAll();
End-if;
```

1. Get a session object.

   Before you can get a Component Interface, you have to get a session object. The session controls access to the Component Interface, provides error tracing, enables you to set the runtime environment, and so on.

   ```
   &MYSESSION = %Session;
   ```

2. Get a Component Interface.

   Use the GetCompIntfc method with a session object to get the Component Interface. You must specify a Component Interface definition that has already been created. You receive a runtime error if you specify a Component Interface that doesn't exist.

   ```
   &MYCI = &MYSESSION.GetCompIntfc(COMPINTFC.BUS_EXP);
   ```

   After you execute the GetCompIntfc method, you have only the *structure* of the Component Interface. You haven't populated the Component Interface with data yet.

3. Set the GETKEYS.

   These are the key values required to return a unique instance of existing data. If the keys you specify allow for more than one instance of the data to be returned, or if no instance of the data matching the key values is found, you receive a runtime error.

```
&MYCI.EMPLID = "8001";
```

4. Get the instance of data for the Component Interface.

   After you set the key values, you must use the Get method.

   ```
   &MYCI.Get();
   ```

   This populates the Component Interface with data, based on the key values you set.

5. Get the level 1 scroll.

   The name of the scroll can be treated like a property. It returns a data collection that contains all the level 1 data.

   ```
   &MYLEVEL1 = &MYCI.BUS_EXPENSE_PER
   ```

6. Get the appropriate item in the level 1 data collection.

   Remember, scroll data is hierarchical. You must get the appropriate level 1 row before you can access the level 2 data.

   ```
   For &I = 1 to &MYLEVEL1.Count
   &ITEM = &MYLEVEL1.Item(&I);
   ```

7. Get the level 2 scroll.

   This is done the same way as you accessed the level 1 scroll, by using the scroll name as a property.

   ```
   &MYLEVEL2 = &ITEM.BUS_EXPENSE_DTL;
   ```

8. Get the appropriate item in the level 2 data collection.

   A data collection is made up of a series of items (rows of data.) You have to access the appropriate item (row) in this level also.

   ```
   &COUNT = &MYLEVEL2.Count
   /* get appropriate item in lvl2 collection */
   For &J = 1 to &COUNT
   &LVL2ITEM = &MYLEVEL2.Item(&J);
   ```

9. Insert or delete a row of data.

   You can insert or delete a row of data from a data collection. The following example finds the last item (row of data) in the second level scroll. If it matches the value of %Date, the last item is deleted. If it doesn't match, a new row is inserted.

   ```
   &CIDATE = &LVL2ITEM.CHARGE_DT;
   If &CIDATE <> %Date Then
      /* insert row */
   &NEWITEM = &MYLEVEL2.InsertItem(&COUNT);
   /* set values for &NEWITEM */
   Else
      /* delete last row */
   &MYLEVEL2.DeleteItem(&COUNT);
   End-If;
   ```

10. Save the data.

When you execute the Save method, the new instance of the data is saved to the database.

```
If NOT(&MYCI.Save()) Then
```

The **Save** method returns a Boolean value: True if the save was successful, False otherwise. Use this value to do error checking.

The standard PeopleSoft save business rules (that is, any PeopleCode programs associated with SaveEdit, SavePreChange, WorkFlow, and so on) are executed. If you did not specify the Component Interface to run in interactive mode, FieldEdit, FieldChange, and so on, also run at this time.

Note. If you're running a Component Interface from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

---

11. Check Errors.

    You can check if there were any errors using the PSMessages property on the session object.

```
If NOT(&MYCI.Save()) Then
   /* save didn't complete */
   &COLL = &SESSION.PSMessages;
   For &I = 1 to &COLL.Count
      &ERROR = &COLL.Item(&I);
      &TEXT = &ERROR.Text;
      /* do error processing */
   End-For;
   &COLL.DeleteAll();
End-if;
```

    If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, delete it from the PSMessages collection.

    The **Text** property of the PSMessage returns the text of the error message. At the end of this text is a contextual string that contains the name of the field that generated the error. The contextual string has the following syntax:

```
{BusinessComponentName.[CollectionName(Row).[CollectionName(Row).⇒
[CollectionName(Row)]]].PropertyName}
```

    For example, if you specified the incorrect format for a date field of the Component Interface named ABS_HIST, the **Text** property contains the following string:

```
Invalid Date {ABS_HIST.BEGIN_DT} (90), (1)
```

    The contextual string (by itself) is available using the **Source** property of the PSMessage.

Note. If you've called the Component Interface from an Application Engine program, all errors are also logged in the Application Engine error log tables.

See Chapter 29, Session Class , Error Handling , page 1560 and Chapter 29, Session Class , Source, page 1574.

CHAPTER 11

Exception Class

This chapter provides an overview of Exception class and discusses the following topics:

- How exceptions are thrown.
- When to use exceptions.
- Try-Catch sections.
- Data Type for Exception class objects.
- Scope of Exception class objects.
- Exception class built-in function and language constructs.
- Exception class methods.
- Exception class properties.

Understanding Exception Class

An *exception* can be defined as any unusual event that requires special handling in your PeopleCode program. This could include some kind of hardware failure, such as trying to write to a printer that's been turned off, or a software failure, such as trying to divide by zero. You can also detect your own errors, and use the **Throw** statement to construct an exception.

Exception handling is the processing you initiate when an exception occurs. You can handle errors in PeopleCode using the **Catch** statement. The places in your code where you want exceptions handled must be enclosed by a **Try** and **End-Try** statement.

The Exception class encapsulates exceptional runtime conditions. It can be used with most PeopleCode programs.

Note. The Exception class does *not* work with any of the PeopleSoft APIs, that is, the classes whose objects are declared as type ApiObject. This includes the Tree classes, the Query classes, Search classes, and so on.

The process of error handling can be broken down as follows:

- An error occurs (either a hardware or software error).
- The error is detected and an exception is thrown (either by the system or by your program).
- Your exception handler provides the response.

How Exceptions Are Thrown

Exceptions get thrown either by the runtime system or by your program.

In general, the kinds of errors that cause the runtime system to throw an exception are errors that terminate your PeopleCode program. For example, dividing by zero, referencing a method or property that doesn't exist, or trying to use a method or property on a null object.

To throw your own exceptions, you can use the **Throw** statement.

When to Use Exceptions

If the error is something that you can easily check, you shouldn't use exceptions. For example, suppose your page had begin and end date fields. In your SavePreChange PeopleCode program, you would want to check to verify that the end date was after the begin date. This kind of error produces incorrect data. It doesn't terminate your PeopleCode program. This is a simple check, one that you don't need to throw an exception for.

Errors that you might want to use exceptions for are the kinds that you are going to check for often. It might make more sense to have a single exception class for that kind of error, so you'd only have to write your error handling code once.

For example, suppose that a function you called was supposed to return an array object that you'd then manipulate using the array class methods and properties. When the function fails and doesn't return an array object, you can either check for Null, or check for an exception (as this kind of error terminates your PeopleCode program.) If you must often check for Nulls, perhaps instead of constantly checking, you could rely on the runtime system throwing exceptions for you.

Pseudo code example:

```
Import ExceptionPackage:*;
/* this package contains all the exception classes for your app */

try
   &MyArray = GetArrayData(&Param1, &Param2, &Param3);
   /* Code to manipulate &MyArray here */
catch ExceptionNull &Ex1
   if &Ex1.MessageSetNumber = 2
      and &Ex1.MessageNumber = 236 Then

/* this is the message set and message number for &MyArray being Null */

   /* do data error handling */

   end-if;
end-try;
```

Considerations Using Exceptions and Transfer Functions

Using functions that transfer the end user to a new component, such as **DoModal**, **DoModalComponent**, or **Transfer** (in some cases) inside a try block do *not* catch PeopleCode exceptions thrown in the new component. Starting a new component starts a brand new PeopleCode evaluation context. Catches are only caught for exceptions thrown within the *current* component.

In the following code example, the catch statement only catches exceptions thrown in the code *prior to* using the DoModal function, but not any exceptions that are thrown within the new component.

```
/* Set up transaction */
If %CompIntfcName = "" Then
   try
      &oTrans = &g_ERMS_TransactionCollection.GetTransactionByName(RB_EM_⇒
WRK.DESCR);
      &sSearchPage = &oTrans.SearchPage;
      &sSearchRecord = &oTrans.SearchRecord;
      &sSearchTitle = &oTrans.GetSearchPageTitle();
      If Not All(&sSearchPage, &sSearchRecord, &sSearchTitle) Then
         Error (MsgGetText(17834, 8081, "Message Not Found"));
      End-If;
      &c_ERMS_SearchTransaction = &oTrans;

      /* Attempt to transfer to hidden search page with configurable filter */
      &nModalReturn = DoModal(@("Page." | &sSearchPage), &sSearchTitle, - 1, - 1);
   catch Exception &e
      Error (MsgGetText(17834, 8082, "Message Not Found"));
   end-try;
```

Try-Catch Sections

The statements that immediately follow the **Try** keyword are called the *protected statements*. These are the only statements that are 'protected' by the catch clauses in the Try-Catch statements. The catch clauses in a Try-Catch statement can be executed only if an exception is thrown by the protected statements. In addition, a catch clause is executed only when handling an exception that matches the type given on the catch. Any exceptions thrown by the catch clauses are not caught by their own Try-Catch statement.

The execution of the Try-Catch statement starts by executing the protected statements. If none of these statements, as well as none of the statements called by them, causes an exception to be thrown, the try-catch is done. The statements in the catch clauses are not executed.

Each catch clause in the Try-Catch section declares a local variable of class Exception or a subclass of Exception. A catch clause local variable has the same scope and lifetime as a local variable declared at the start of the catch clause.

The following is the general order of exception execution and handling in the Try-Catch statements.

1. The exception is thrown.
2. The exception is considered by the first enclosing Try-Catch statement.

Try-catch statements can be nested.

This is a dynamic enclosure, that is, tracking back through method and function callers. Any intervening non-Try-Catch statements are skipped.

3. Within this Try-Catch statement, the catch clauses are considered in program text order, that is, first the first catch clause is considered, then the next, and so on to the last. At each consideration, the type (class) of the exception being thrown is compared to the type of the local variable declared by the Catch clause. If the exception being thrown is the same as the Catch type, or is a subtype of it (so the exception could be assigned to the Catch's local variable) the Catch clause matches the exception being thrown.
4. If a matching catch clause is found, that catch clause handles the exception. The exception is considered 'caught', which means that it does not propagate further. Execution proceeds by assigning the thrown object to the catch clause's local variable and executing the statements in the catch clause. When one of the catch clauses handles the exception, (and the statements in the catch clause don't throw any further exceptions), execution continues normally after the Try-Catch statement.

   If a statement in the catch clause throws another exception, that exception begins to propagate from the point of the throw.
5. If a matching catch clause is not found, that is, if no catch clauses can accept the thrown object, then the thrown object is still uncaught. The thrown object is considered by the next dynamically enclosing Try-Catch statement (return to step 3).
6. If a thrown object is not caught by any catch clause in any dynamically enclosing Try-Catch statements, a fatal PeopleCode evaluation error is produced.

Data Type for Exception Class Objects

Exception Class objects are declared as type Exception. For example:

```
Local Exception &Ex1;
```

Scope of Exception Class Objects

Exception objects can be instantiated only from PeopleCode. This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

Exception Class Built-in Functions

Chapter 1, PeopleCode Built-in Functions, CreateException , page 97

Chapter 1, PeopleCode Built-in Functions, Throw , page 457

Chapter 1, PeopleCode Built-in Functions, Try , page 482

Exception Class Methods

In this section, we discuss the Exception class methods, in alphabetical order.

GetSubstitution

Syntax

GetSubstitution(*Index*)

Description

When you create a message in the message catalog, you can specify values in the message text, such as %1, %2, and so on, that are replaced with values by the system at runtime. The value that gets put into the message text value is called the *substitution string*. The **GetSubstitution** method returns the substitution string in the error message specified by *Index*, with 1 being the first substitution string, 2 being the second, and so on.

Parameters

Parameter	Description
Index	Specify the substitution string number that you'd like to return. The first one is 1, the second is 2, and so on. You must specify a valid substitution string number, that is, you can't specify a 4 if there are only three substitution strings.

Returns

The substitution text as a string.

Example

```
If &Ex1.MessageSetNumber = 2 and &Ex1.MessageNumber = 170 Then
   /* Get Sendmail error */
      &String = &Ex1.GetSubstitution(1);
   /* do processing according to type of Sendmail error */
   . . . .
End-if;
```

See Also

Chapter 11, Exception Class , SetSubstitution, page 840, *PeopleSoft Server Tools Administration*, PeopleTools Utilities, Message Catalog.

Output

Syntax

Output()

Description

Use the **Output** method if you want to display the message associated with the exception to the user (in an online context), or have it logged (if offline). This is analogous to using the **MessageBox** function to log a message.

Parameters

None.

Returns

If called in an online context, display the message associated with the exception to the user. If called offline, record the exception for later display or analysis.

Example

```
catch Except2 &Ex2
   &Ex2.Output(); /* tell about it */
```

See Also

Chapter 1, PeopleCode Built-in Functions, Throw , page 457.

SetSubstitution

Syntax

SetSubstitution(*Index, String*)

Description

When you create a message in the message catalog, you can specify values in the message text, such as %1, %2, and so on, that are replaced with values by the system at runtime. The value that gets put into the message text value is called the *substitution string*. The **SetSubstitution** method sets the substitution string in the error message specified by *Index*, with 1 being the first substitution string, 2 being the second, and so on.

Parameters

Parameter	Description
Index	Specify the substitution string number that you'd like to replace. The first one is 1, the second is 2, and so on. You must specify a valid substitution string number, that is, you can't specify a 4 if there are only three substitution strings.
String	Specify the text of the substitution string.

Returns

None.

See Also

Chapter 11, Exception Class , GetSubstitution, page 839.
PeopleSoft Server Tools Administration, PeopleTools Utilities, Message Catalog

ToString

Syntax

ToString()

Description

The **ToString** method returns the expanded message text associated with the exception in the current user's current language. The string returned is the message text (in the current user's language) with the substitution string replacing the substitution markers (%1 and so on) in the text.

Parameters

None.

Returns

A string.

See Also

Chapter 11, Exception Class , Output, page 839.

Exception Class Properties

In this section, we discuss the Exception class properties, in alphabetical order.

Context

Description

This property returns a string description of the location of the condition in PeopleCode or other processing. This might contain a stack backtrace at the point of the exceptional condition.

This property is read-only.

Example

This is the "At" part of the error message display. For example, for an exception thrown in Record.Field Event QE_31DIGREC1.QE_31DIGFLD5 FieldChange function EachComp at PeopleCode program counter 671, statement 11 of the source program, the Context would be:

```
At QE_31DIGREC1.QE_31DIGFLD5.FieldChange EachComp   PCPC:671  Statement:11
```

DefaultText

Description

This property sets a string to use as the basic message text for the exception when the message can't be found in the message catalog.

This property is read-write.

MessageNumber

Description

This property is the message number for a message describing the exceptional condition.

This property is read-only.

MessageSetNumber

Description

This property is the message set number for a message describing the exceptional condition.

This property is read-only.

MessageSeverity

Description

This property is the message severity as a string for a message describing the exceptional condition. The message severity is set for the message when it's created in the message catalog. The values are:

Numeric Value	Constant Value	Description
C	%MsgSeverity_Cancel	Message severity is Cancel.
E	%MsgSeverity_Error	Message severity is Error.
M	%MsgSeverity_Message	Message severity is Message.
W	%MsgSeverity_Warning	Message severity is Warning.

This property is read-only.

SubstitutionCount

Description

This property returns the number of substitution strings in the message text as a number.

This property is read-only.

CHAPTER 12

Field Class

This chapter provides an overview of Field class and discusses the following topics:

- Shortcut considerations
- Data Type for a field object
- Scope of a field object
- Field class built-in function
- Field class methods
- Field class properties

Understanding the Field Class

A *field* object, instantiated from the Field class, is a single instance of data within a record object, and is based on a field definition.

Accessing or changing the value of a field using the **Value** property is a common action.

SetDefault is a frequently used method. **Name, Enabled,** and **Type** are several commonly used field properties. If you are working with message objects, **EditError** is also a commonly used field property.

The Field class is one of the data buffer access classes.

See Also

PeopleCode Developer's Guide, Data Buffer Access

Shortcut Considerations

An expression of the form

```
FIELD.fieldname.property
```

or

```
FIELD.fieldname.method(. . .)
```

is converted to an object expression by using **GetField**(FIELD.*fieldname*). For example, the next two lines of code are identical:

```
FIELD.CHECKLIST_DT.Enabled = False;

GetField(FIELD.CHECKLIST_DT).Enabled = False;
```

An expression of the form

```
recname.fieldname.property
```

or

```
recname.fieldname.method(.  .  .)
```

is converted to an object expression by using **GetField**(*recname.fieldname*). For example, the next two lines of code are identical:

```
EMPL_CHECKLINST.CHECKLIST_DT.Enabled = False;

GetField(EMPL_CHECKLIST.CHECKLIST_DT).Enabled = False;
```

Data Type for a Field Object

Field objects are declared as type Field. For example,

```
Local Field &MyField;
```

Scope of a Field Object

A field can only be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

Field Class Built-in Function

Chapter 1, PeopleCode Built-in Functions, GetField , page 242

Field Class Methods

In this section, we discuss the Field class methods. The methods are discussed in alphabetical order.

AddDropDownItem

Syntax

AddDropDownItem(*CodeString*, *DescriptionString*)

Description

The **AddDropDownItem** method adds an item to the dropdown list in the control for the field. The first time this method is called, it overrides the prompt table or translate table used to populate the list. Those items no longer appear in the list. Only the items added using this method display.

Subsequent calls to this method *adds* additional items to the dropdown list. The items added with the first call to the method also display.

If there is an existing value and the dropdown list is changed with these functions, the selection shows as (Invalid value) unless the new list contains an entry with the same code as the existing value.

Considerations Using AddDropDownItem

If the data for the dropdown is language sensitive, the values for the dropdown should come from the message catalog or from a database field that has a related language record, and should *not* be hard-coded.

A good place for your PeopleCode program to populate a dropdown list is in the RowInit event. This event executes before the page is shown for the first time, so it prevents unnecessary SQL.

Parameters

Parameter	Description
CodeString	Specify the value used to set the field value if this item is selected. Codes longer than the size of the field are truncated.
DescriptionString	Specify the value the end-user sees in the dropdown list.

Returns

None.

Example

Using a hardcoded list is not appropriate for this function because translations do not work. The data must come from the Translate Table (or other record) directly so that the data is translated correctly.

```
Local Rowset &Xlat;

&FLD = GetRecord(Record.JOB).GetField(Field.ACTION);
&FLD.ClearDropDownList();

Evaluate %Component
When Component.JOB_DATA_CONCUR
   &Xlat = CreateRowset(Record.PSXLATITEM);
   &Xlat.Fill("WHERE FILL.FIELDNAME = 'ACTION' AND Fill.FIELDVALUE in ⇒
('ADL','HIR') and EFFDT = (select max(EFFDT) from PSXLATITEM B where B.FIELDNAME =⇒
 'ACTION' and B.FIELDVALUE in ('ADL','HIR') and EFFDT <= JOB.EFFDT)");
```

```
      &Xlat_cnt = &Xlat.ActiveRowCount;
      For &I = 1 To &Xlat_cnt
         &CodeIn = &Xlat.GetRow(&I).GetRecord(1).FIELDVALUE.Value;
         &DescIn = &Xlat.GetRow(&I).GetRecord(1).XLATLONGNAME.Value;

         &FLD.AddDropDownItem(&CodeIn, &DescIn);
      End-For;

      Break;
   When-Other
   End-Evaluate;
```

See Also

Chapter 12, Field Class , ClearDropDownList , page 846.

ClearDropDownList

Syntax

ClearDropDownList()

Description

The **ClearDropDownList** method clears all items added to the dropdown list using the AddDropDownItem method. In addition, this method causes the prompt table or translate table values defined for the list to come back into effect again (unless they're subsequently overridden again with AddDropDownItem.)

Parameters

None.

Returns

None.

Example

```
&FLD = GetRecord(Record.ABSENCE_HIST).GetField(Field.ABSENCE_TYPE);
&FLD.AddDropDownItem("CNF", "Conference");
&FLD.AddDropDownItem("VAC", "Vacation");
&FLD.AddDropDownItem("SCK", "Sick");
. . . .
&FLD.ClearDropDownList();
```

See Also

Chapter 12, Field Class , AddDropDownItem , page 845.

GetAuxFlag

Syntax

GetAuxFlag(*FlagNumber*)

Description

Use the **GetAuxFlag** method to determine whether the Auxiliary Flag Mask specified by *FlagNumber* has been set for a field.

Currently, only one flag comes preset from PeopleSoft: a 1 indicates a ChartField.

Parameters

Parameter	Description
FlagNumber	Specify the flag number. 1 is a ChartField. Valid values are between 1 and 32.

Returns

A Boolean value: True, the flag is set, False if the flag hasn't been set.

Example

```
&field = GetField(Field.RC_TEST_PB);
&ret = &field.GetAuxFlag(1);
If (&ret = True) Then
   MessageBox(0, "Metadata Fn Status", 0, 0, "Aux Flag 1 is SET - ChartField");
Else
   MessageBox(0, "Metadata Fn Status", 0, 0, "Aux Flag 1 is NOT SET - Not a Chart⇒
Field");
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetDBFieldAuxFlag , page 405.

GetLongLabel

Syntax

GetLongLabel(*LabelID*)

Description

The **GetLongLabel** method returns the long name for a field given a label ID. If the given label ID isn't found, a null string is returned. *LabelID* takes a string value.

Note. If a hyperlink or button has an associated record field, and a custom label has been set for that field using PeopleCode, this label value is used for the ToolTip pop-up in place of the default long record field text.

Returns

A text string containing the long name of the field for the specified label ID.

Example

The following code sets the label for a field to one of two different texts, based on the page.

```
Local Field &FIELD;
```

```
&FIELD = GetField(RECORD.MYFIELD);
   If %Page = PAGE.PAGE1 Then
      &FIELD.Label = &FIELD.GetLongLabel("LABEL1");
   Else If %Page = PAGE.PAGE2 Then
      &FIELD.Label = &FIELD.GetLongLabel("LABEL2");
End-If;
```

If the Label ID is the same as the name of the field, you could use the following:

```
&LABELID = &FIELD.Name;
&FIELD.Label = &FIELD.GetLongLabel(&LABELID);
```

See Also

Chapter 12, Field Class , GetShortLabel, page 849, Chapter 12, Field Class , Label, page 861, Chapter 1, PeopleCode Built-in Functions, SetLabel , page 419.

GetRelated

Syntax

GetRelated(*recname.fieldname*)

Description

The **GetRelated** method returns a field object for a related field *recname.fieldname* that has the field executing the method as its control field.

This method is similar to the **GetRelField** built-in function, however, the built-in works only in the current context, while this method can be applied to a field from any position in the buffer structure.

Using GetRelated With a Control Field

PeopleCode events on the Control Field can be triggered by the Related Edit field. When this happens, there can be different behavior than with other types of fields:

- If the events are called from FieldEdit of the Control Field, and that FieldEdit is triggered by a change in the Related Edit field, the functions return the previous value.
- If the events are called from FieldChange of the Control Field, and that FieldChange is triggered by a change in the Related Edit field, the functions return the value entered into the Related Edit. This may be a partial value, that will subsequently be expanded to a complete value when the processing is complete.

Parameters

Parameter	Description
recname.fieldname	Specifies a field in the same row as the current field object, that has the field executing the method as its control field.

Returns

The field object for the field with the specified name that is related to the field executing the method.

Example

In the following example, the field object is instantiated, then the related display field object. The **Value** property for the related display is changed, it is disabled, and another variable is assigned its value.

```
Local Field &FIELD, &REL_FIELD;

&FIELD = GetField(OPC_9A2FIELDS.COMPANY);
/* control field object  */
&REL_FIELD = &FIELD.GetRelated(COMPANY_TBL.DESCR);
/* related display object */
&REL_FIELD.Value = "Change";
&REL_FIELD.Enabled = False;
&TMP = &REL_FIELD.Name;
&REL_FIELD.Value = &TMP;
```

If you were not going to use the &FIELD variable later, the first two lines of code in the previous example could be combined:

```
&REL_FIELD = GetField(OPC_9A2FIELDS.COMPANY).GetRelated(COMPANY_TBL.DESCR);
```

Suppose you had two control fields, EMPLID and MANAGER_ID, and both use the NAME field on the PERSONAL_DATA table as their related display. If you need to access both related display fields, you could do the following:

```
&NAME_EMPLID = GetField(PERSONAL_DATA.EMPLID).GetRelated(PERSONAL_DATA.NAME);

&NAME_MANAGER = GetField(PERSONAL_DATA.MANAGER_ID).GetRelated(PERSONAL_DATA.NAME);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetField , page 242.

GetShortLabel

Syntax

GetShortLabel(*LabelID*)

Description

The **GetShortLabel** method returns the short name for a field given a label ID. If the given label ID isn't found, a null string is returned. *LabelID* takes a string value.

Note. If a hyperlink or button has an associated record field, and a custom label has been set for that field using PeopleCode, this label value is used for the ToolTip pop-up in place of the default long record field text.

Returns

A text string containing the short name of the field for the specified label ID.

Example

The following code sets the label for a field to one of two different texts, based on the page.

```
Local Field &FIELD;
```

```
&FIELD = GetField(RECORD.MYFIELD);
   If %Page = PAGE.PAGE1 Then
      &FIELD.Label = &FIELD.GetShortLabel("LABEL1");
   Else If %Page = PAGE.PAGE2 Then
      &FIELD.Label = &FIELD.GetShortLabel("LABEL2");
End-If;
```

If the Label ID is the same as the name of the field, you could use the following:

```
&LABELID = &FIELD.Name;
&FIELD.Label = &FIELD.GetShortLabel(&LABELID);
```

See Also

Chapter 12, Field Class , GetLongLabel, page 847, Chapter 12, Field Class , Label, page 861, Chapter 1, PeopleCode Built-in Functions, SetLabel , page 419.

SearchClear

Syntax

SearchClear()

Description

The **SearchClear** method clears the field values if the field is a search key.

Considerations Using SearchClear and SetDefault

SetDefault causes a field to be set to its default value (if there is one) immediately after SearchInit PeopleCode finishes. SetDefault overrides SearchClear. If you call SearchClear for a record, then use SetDefault for a field, the field is set to its default value and the search key values for the rest of the record are cleared.

Parameters

None.

Returns

None.

Example

```
Local Field &FIELD;

&FIELD = GetField(PERSONAL_DATA.EMPLID);
&FIELD.SearchClear();
```

See Also

Chapter 12, Field Class , SetDefault, page 851, Chapter 25, Record Class , SearchClear, page 1453.

SetCursorPos

Syntax

SetCursorPos(**PAGE**.*pagename* | **%Page**)

Description

The **SetCursorPos** enables you to set the focus to the page field corresponding to the field object (on the specified page). The current page may be specified as %Page.

Restrictions on Use with a Component Interface

This function is ignored (has no effect) when used by a PeopleCode program that's been called by a Component Interface.

Parameters

Parameter	Description
Pagename	The name of the page, preceded by the keyword **Page**. The *pagename* page must be in the current component. You can also pass the %page system variable in this parameter (without the **Page** reserved word).

Returns

None.

Example

The following pseudo-code enables you to set the focus to a related field:

```
GetField(ControlRec.ControlField).GetRelated(RelatedRec.RelatedField).⇒
SetCursorPos(PAGE.pagename);
```

See Also

Chapter 12, Field Class , SetDefault, page 851, Chapter 1, PeopleCode Built-in Functions, SetCursorPos , page 404.

SetDefault

Syntax

SetDefault()

Description

SetDefault sets the value of the field to a null value, or to a default, depending on the type of field.

- If this method is used against data from the Component buffers, the next time default processing occurs, it is set to its default value: either a default specified in its record field definition or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.
- If this method is used with a field that isn't part of the data buffer (for example, a field in a record object instantiated with CreateRecord) the field is automatically set to its default value if one is set for the **field**, not

for the record field. Any FieldDefault PeopleCode will *not* be run on these types of fields. If you want to set the default values for all the fields in a record, use the **SetDefault** record class method.

Blank numbers correspond to zero on the database. Blank characters correspond to a space on the database. Blank dates and long characters correspond to NULL on the database. **SetDefault** gives each field data type its proper value.

Parameters

None.

Returns

None.

Example

```
&CHARACTER.SetDefault();
```

See Also

Chapter 25, Record Class , SetDefault, page 1456, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Default Processing.

Field Class Properties

In this section, we discuss the Field class properties. The properties are discussed in alphabetical order.

DataAreaCollapsed

Description

This property specifies whether the default initial view of the data area of a group box is collapsed or expanded.

Note. You must set the Collapsible Data Area field on the properties for the group box for this property to have any effect.

This property changes to reflect the current state of the data area, according to whether the user has collapsed or expanded it. Changing the value collapses or expands the data area, but it does *not* prevent the user from collapsing (or expanding) it themselves.

Note. Because the user can change the value of this property, whatever value is set in PeopleCode isn't guaranteed to be still set the next time it is checked, because the user may have collapsed or expanded the data area in the meantime.

This property overwrites the value of the Default Initial View to Expandedfield set in PeopleSoft Application Designer. For example, if Default Initial View to Expanded is selected in PeopleSoft Application Designer, then the value for the DataAreaCollapsed property is set to True, the control initially displays collapsed.

This property takes a Boolean value: True, initially display the data area collapsed, False, initially display the data area expanded.

This property is read-write.

Note. If you want to collapse an entire level-based control, such as a scroll area or a grid, use the DataAreaCollapsed Rowset method.

See Also

Chapter 27, Rowset Class , DataAreaCollapsed, page 1503.

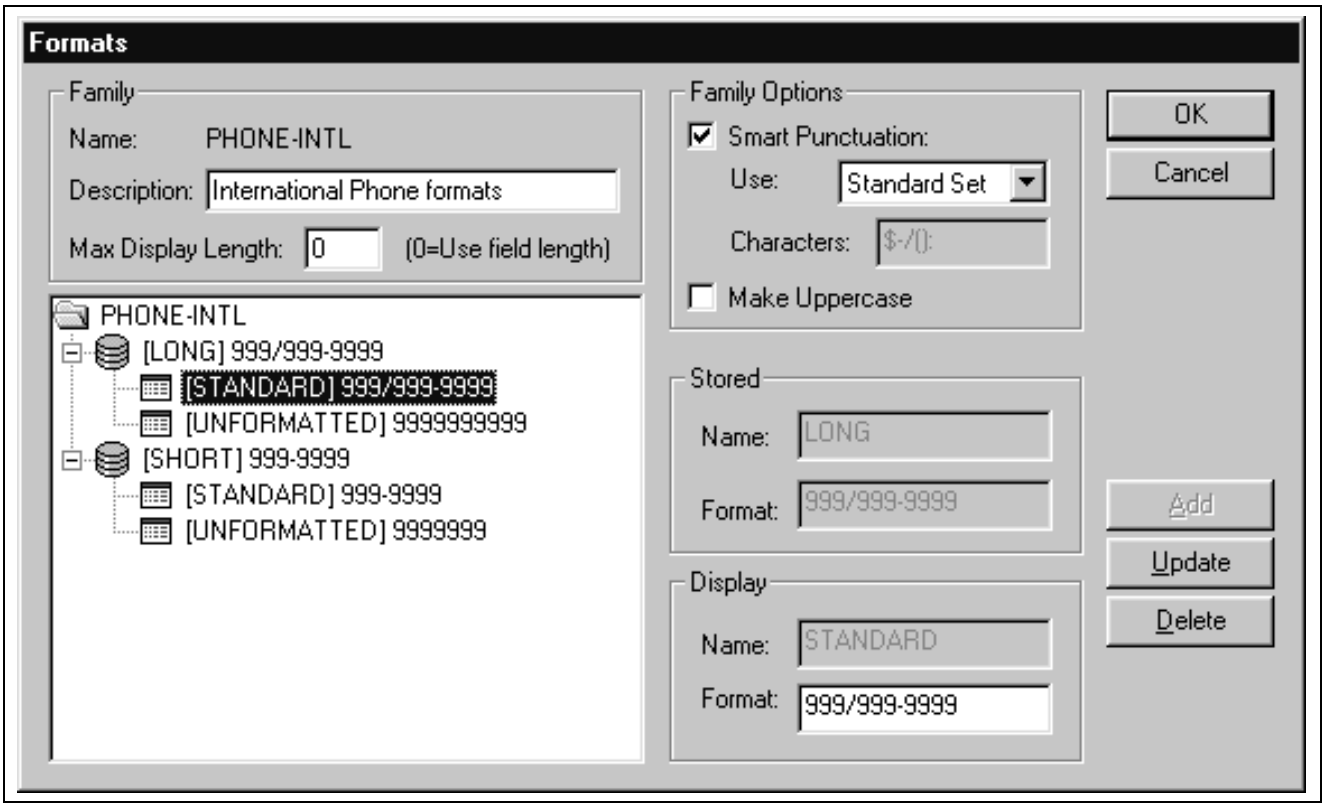
DisplayFormat

Description

Use this property to specify a custom format to use for the field.

This property is read-write.

The custom format for a field is specified in the field definition. This property enables you to switch between display formats that are defined as part of a custom format. For example, suppose your field used the PHONE-INTL custom format:



PHONE-INTL format

Both the LONG and the SHORT stored formats have two display formats: STANDARD and UNFORMATTED.

Using this property, you could select either of the display formats. For example:

```
If %Component = PAGE.INTERNATIONAL
   &CHAR.DisplayFormat = "STANDARD";
Else
```

```
   &CHAR.DisplayFormat = "UNFORMATTED";
End-If;
```

Note. You *cannot* change the Stored format, but you can find its value using the StoredFormat property.

Example

```
&CHARACTER.DisplayFormat = "STANDARD";

/* this assumes that &CHARACTER is a custom formatted field, and a display format⇒
 option is STANDARD */
```

See Also

Chapter 12, Field Class , StoredFormat, page 868.

DisplayOnly

Description

This property is set to True if the field is set to DisplayOnly. May be set to False to change how the field displays.

Note. This property *overwrites* whatever value is set in PeopleSoft Application Designer.

This property is read-write.

EditError

Description

This property is True if an error for this field has been found after executing the **ExecuteEdits** method with either a message object or a record object. This property can be used with the **MessageSetNumber** and **MessageNumber** properties to find the error message set number and error message number.

This property is read-write. After you have fixed the errors, you must set this property to False before running **ExecuteEdits** again.

Example

The following is an example showing how **EditError**, along with the **ExecuteEdits** method could be used:

```
&REC.ExecuteEdits();
If &REC.IsEditError Then
   For &I = 1 to &REC.FieldCount
      If &REC.GetField(&I).EditError Then
         LOG_ERROR();  /* application specific pgm */
      End-If;
   End-For;
End-If;
```

See Also

Chapter 25, Record Class , ExecuteEdits, page 1448, Chapter 17, Message Class , ExecuteEdits, page 987, Chapter 12, Field Class , MessageNumber, page 863, Chapter 12, Field Class , MessageSetNumber, page 863.

Enabled

Description

This property is True if this field is enabled. May be set False to disable any control that displays this field.

This property is read-write.

Example

```
&CHARACTER.Enabled = True;

&CHARACTER.Enabled = False;
```

FieldLength

Description

This property returns the length of the field as a number.

This property is read-only.

Example

```
&MyRec = CreateRecord(Record.BKREC1);
&MyField = &MyRec.GetField(Field.BKNAME);
&length = &MyField.FieldLength;
MessageBox(0, "Field Length", 0, 0, "The field BKREC1.BKNAME is length " |⇒
 &length);
```

FormatLength

Description

This property returns the length of the format for the field as a number.

This property is read-only.

Example

```
&field = GetField(Field.BKTEST);
&formatlength = &field.FormatLength;
&length = &field.FieldLength;
&mymsg = "Field.BKTEST length is " | &length | " and format length is " |⇒
 &formatlength;
MessageBox(0, "MetaData Fn Status", 0, 0, &mymsg);
```

FormattedValue

Description

This property returns the value of a field as a string, formatted exactly as it would be displayed on an edit field on a page. This is useful when you're using a prompt field for the label of another field.

Because a record field can be bound to more than one page field, FormattedValue takes the first associated page field, looking first on the current page, then through all the pages in the component. This property returns a null string ("") if used with an Image or ContentReference field. It also returns a null string if no associated page field is found.

This property is read-write.

Example

This property could be used to set up a hyperlink labeled by data generated on a page. To do this, do the following:

1. Define a work field.
2. Associate the work field with the hyperlink page field.
3. Define a hidden page field with the formatting that you want.
4. Associate the hidden page field with the data field.
5. Write PeopleCode (probably for RowInit) to set the label on the work field to be the FormattedValue of the data field.

For example, suppose you wanted a hyperlink labeled by the QE_POSITION_ID field on the QE_PLYR_POSITN record. This field is a prompt table field, edited by table QE_POSITION.

You want to display the DESCRSHORT field from the prompt table, instead of the QE_POSITION_ID code value. Create a work field, POS_LINK on the WORK_REC record, and create a hyperlink page field associated with the DESCRSHORT field. For the label, create an invisible drop-down list page field, associated with QE_PLYR_POSITN.QE_POSITION_ID. Set its Prompt Table Field to DESCRSHORT. Then put a PeopleCode program on the component RowInit for either the QE_PLYR_POSITN record or the WORK_REC.

```
WORKREC.POS_LINK.Label = QE_PLYR_POSITN.QE_POSITION_ID.FormattedValue;
```

IsAltKey

Description

This property is True if this field references a field definition that is defined as an Alternate Search key in the associated record definition.

This property is read-only.

IsAuditFieldAdd

Description

This property is True if this field references a field definition that is defined as an audit field in the associated record definition, and the field is audited whenever new data is added.

This property is read-only.

IsAuditFieldChg

Description

This property is True if this field references a field definition that is defined as an audit field in the associated record definition, and the field is audited whenever data is changed.

This property is read-only.

IsAuditFieldDel

Description

This property is True if this field references a field definition that is defined as an audit field in the associated record definition, and the field is audited whenever data is deleted.

This property is read-only.

IsAutoUpdate

Description

This property is True if this field references a field definition that is defined as an Auto-Update field in the associated record definition.

This property is read-only.

IsChanged

Description

This property returns True if the value for the field has been changed.

Note. Use this property with the primary database record only. It will not return valid results if used with a work record.

This property is read-only.

Example

```
If &CHARACTER.IsChanged Then

    Warning ("The character field has been changed");

End-If;
```

IsDateRangeEdit

Description

This property is True if this field references a field definition that is defined as requiring a Reasonable Date in the associated record definition.

This property is read-only.

IsDescKey

Description

This property is True if this field references a field definition that is defined as a descending key in the associated record definition.

This property is read-only.

IsDuplKey

Description

This property is True if this field references a field definition that is defined as a duplicate key in the associated record definition.

This property is read-only.

IsEditTable

Description

This property is True if this field references a field definition that has a table edit type of a Prompt Table with Edit.

This property is read-only.

IsEditXlat

Description

This property is True if this field references a field definition that has a table edit type of a Translate Table Edit.

This property is read-only.

IsFromSearchField

Description

This property is True if this field references a field definition that is defined as a From Search Field in the associated record definition.

This property is read-only.

IsInBuf

Description

This property returns True if the data of the field is present in the data buffers. For example, all of the fields of a derived record are not present in the data buffer.

This property is read-only.

Example

The following example iterates over all the fields in a record. The code verifies that the data for the field is accessible before it tries to assign the value to a variable.

```
For &I = 1 to &REC.FieldCount
   &FIELD = &REC.GetField(&I);
   If &FIELD.IsInBuf Then
      &VALUE = &FIELD.Value;
      /* do other processing */
   End-If;
End-For;
```

See Also

PeopleCode Developer's Guide, Referencing Data in the Component Buffer, What Record Fields Are in the Component Buffer?.

IsKey

Description

This property is True if this field references a field definition that is defined as a primary key of the associated record definition.

This property is read-only.

Example

```
If &CHARACTER.IsKey Then

    Warning ("The field " | &Character.Name | " is a key");

End-If;
```

IsListItem

Description

This property is True if this field references a field definition that is defined as a List Box item of the associated record definition.

This property is read-only.

IsNotUsed

Description

This property is True if this field has been set as NotUsed with the **SetDBFieldNotUsed** function.

This property is read-only.

Example

```
&field = GetField(Field.RC_TEST_PB);
```

```
&ret = &field.IsNotUsed;
If (&ret = True) Then
   MessageBox(0, "MetaData Fn Status", 0, 0, "Field is NOTUSED");
Else
   MessageBox(0, "MetaData Fn Status", 0, 0, "Field is used.");
End-If;
```

IsRequired

Description

This property is True if this field references a field definition that is defined as Required in the associated record definition.

This property is read-only.

IsSearchItem

Description

This property is True if this field references a field definition that is defined as a Search key in the associated record definition.

This property is read-only.

IsSystem

Description

This property is True if this field references a field definition that is defined as System Maintained field in the associated record definition.

This property is read-only.

IsThroughSearchField

Description

This property is True if this field references a field definition that is defined as a Through Search Field in the associated record definition.

This property is read-only.

IsUseDefaultLabel

Description

This property is True if this field references a field definition that has its label defined as Use Default Label in the associated record definition.

This property is read-only.

IsYesNo

Description

This property is True if this field references a field definition that has a table edit type of a Yes/No Table Edit.

This property is read-only.

Label

Description

This property returns a reference to the label you have set for the field. You can use this property to change the label text for the field when it's displayed on a page.

This property returns a blank string if you use it before you've set the label with either the SetLabel function or with this property.

To return the text of a label before you set it, use the GetShortLabel or GetLongLabel methods.

Note. You can't use this property to set labels longer than 100 characters. If you try to set a label of more than 100 characters, the label is truncated to 100 characters.

The "tool tip," or mouse over text, that appears with a hyperlink at runtime comes from the RFT long label assigned to the record field. However, the RFT long label displays only if it is different from the assigned display value of the hyperlink and it is not null. If the link is an image button, the tool tip is derived from the label text if there is any. Otherwise, the RFT long label is used.

This property is read-write.

Example

The following code sample changes all the field labels for a record to the short label. It assumes that there is a label with the same name as the name of the field for all fields in the record.

```
Local Field &FLD;
Local Record &REC;

If CHECK_FIELD Then
   &REC = GetRecord();
   For &I = 1 to &REC.FieldCount
      &FLD = &REC.GetField(&I);
      &LABELID = &FLD.Name;
      &FLD.Label = &FLD.GetShortLabel(&LABELID);
   End-For;
End-If;
```

See Also

Chapter 12, Field Class , GetShortLabel, page 849, Chapter 12, Field Class , GetLongLabel, page 847.

LabelImage

Description

Use this property to set the image for a pushbutton. This property is only valid for pushbutton controls. This property can be set even when the pushbutton is disabled or display-only.

This property only *overrides* the existing image. It doesn't *change* it. The next time the page is accessed, the original image is used.

The value for this property can be either of the following:

```
Image.imagename
```

or

```
"Imagename"
```

PeopleSoft strongly recommends that if the PeopleCode program sets the LabelImage for an active pushbutton, it also sets the image appropriately if the pushbutton is disabled.

This property is read-write.

Example

```
Local Field &MyField;

&MyField = GetField(BUTTON_BOTTOM);
&MyField.LabelImage = Image.BTN_REFRESH;
```

LongTranslateValue

Description

This property returns a string that contains the Long translate (XLAT) value of the field if the field is based on a translate table.

If the field has a null value, a null string is returned. If the field isn't based on a translate table, or the value isn't in the translate table, the field's current value is returned. Because the current value can be of any type, this property has a type of Any.

Note. If you're accessing a field based on the Translate table, the Value property returns only the one or two letter XLAT value.

Use the ShortTranslateValue property to return the short translate value of a field.

This property is read-only.

Example

```
Local Any &VALUE;
Local Field &MYFIELD;

&MYFIELD = GetField();
&VALUE = &MYFIELD.LongTranslateValue;
```

```
If ALL(&VALUE) Then
   /* do processing */
End-if;
```

See Also

Chapter 12, Field Class , Value, page 869, Chapter 12, Field Class , ShortTranslateValue, page 866.

MessageNumber

Description

This property returns the error message number (as a number) if an error for this field is found after executing **ExecuteEdits** method. You can use this property in conjunction with the **EditError** property (which can be used to determine whether there are any errors) and the **MessageSetNumber** property (which contains the error message set number if an error is found.)

This property is read-only.

Example

```
For &I = 1 to &RECORD.FieldCount
   &MYFIELD = &RECORD.GetField(&I);
   If &MYFIELD.EditError Then
      &MSGNUM = &MYFIELD.MessageNumber;
      &MSGSET = &MYFIELD.MessageSetNumber;
      /* Do processing */
   End-If;
End-For;
```

MessageSetNumber

Description

This property returns the error message set number (as a number) if an error for this field is found after executing **ExecuteEdits** method. You can use this property in conjunction with the **EditError** property (which can be used to determine whether there are any errors) and the **MessageNumber** property (which contains the error message number if an error is found.)

This property is read-only.

Example

```
For &I = 1 to &RECORD.FieldCount
   &MYFIELD = &RECORD.GetField(&I);
   If &MYFIELD.EditError Then
      &MSGNUM = &MYFIELD.MessageNumber;
      &MSGSET = &MYFIELD.MessageSetNumber;
      /* Do processing */
   End-If;
End-For;
```

Name

Description

This property returns the name of the field definition that the field object is based on as a string value.

This property is read-only.

Example

```
WinMessage("The character field's name is : " | &CHARACTER.Name);
```

OriginalValue

Description

This property returns the value of a field, that is, the value that from the database. If the value hasn't been changed and saved by the user, it is the original value from the database. If the value has been changed and saved by the user, it is the existing value in the database.

Note. This property does *not* work for derived records. Original values are the database values, and derived records do not have a corresponding database value.

This property is read-only.

Example

```
&Orig = &MyField.OriginalValue;
If &Orig = &Date Then
   /* do current day processing */
Else
   /* do other processing */
End-If;
```

ParentRecord

Description

This property returns a reference to the record object for the record containing the field.

This property is read-only.

Example

```
&NUMBER_OF_FIELDS = &CHARACTER.ParentRecord.Fieldcount;

/* note that FieldCount is a property of the Record class */
```

PromptTableName

Description

This property returns the name of the prompt table (if any) associated with this field. This property returns a string value. This property returns Null if no prompt table is associated with the field.

This property is read-only.

SearchDefault

Description

If this property is set to True, system defaults (the default values set in the record field definitions) are enabled on search dialogs for the field. Setting this property to True *does not* cause the FieldDefault event to fire.

The system default is done only once, when the search dialog first starts, immediately after any SearchInit PeopleCode. If the user subsequently blanks out a field, the field isn't reset to the default value. Setting SearchDefault to False disables default processing for the field executing the property.

SearchDefault is effective only when used in SearchInit PeopleCode programs.

This property is read-write.

Example

```
&CHARACTER.SearchDefault = True;

/* assuming &CHARACTER is a search key, and has a default value */
```

This example turns on edits and system defaults for the SETID field in the search dialog box:

```
&SETID = GetField(INV_ITEMS.SETID);
&SETID.SeachDefault = True;
&SETID.SearchEdit = True;
```

SearchEdit

Description

If this property is set to True, **SearchEdit** enables system edits (edits specified in the record field definition) for the field, for the life of the search dialog box. Setting SearchDefault to False disables system edits. In the Add mode search dialog box, the following edits are performed when the end-user clicks the Add button. In any other mode, the following edits are performed when the end-user clicks the Search button:

- Formatting
- Required Field
- Yes/No Table
- 1/0 Table
- Translate Table
- Prompt Table

SearchEdit does not cause the FieldEdit, FieldChange, or SaveEdit PeopleCode events to fire during the search dialog.

You might use SearchEdit to control access to the system. For example, you can apply this function to the SETID field of a dialog box and require the user to enter a valid SETID before they are able to click OK on the search dialog box.

This property is read-write.

Considerations Using SearchEdit

If you use this method in the SearchInit event, the search page options are limited to the "=" and "IN" operators.

Example

```
&CHARCTER.SearchEdit = True;

/* assuming &CHARACTER is a search key, and contains an edit table such as⇒
 translate table defined in its field properties.  */
```

This example turns on edits and system defaults for the SETID field in the search dialog box:

```
&SETID = GetField(INV_ITEMS.SETID);
&SETID.SeachDefault = True;
&SETID.SearchEdit = True;
```

ShortTranslateValue

Description

This property returns a string that contains the Short translate (XLAT) value of the field if the field is based on a translate table.

If the field has a null value, a null string is returned. If the field isn't based on a translate table, or the value isn't in the translate table, the field's current value is returned. Because the current value can be of any type, this property has a type of Any.

Note. If you're accessing a field based on the translate table, the Value property returns only the one or two letter XLAT value.

Use the LongTranslateValue property to return the long translate value of a field.

This property is read-only.

Example

```
Local Any &VALUE;
Local Field &MYFIELD;

&MYFIELD = GetField();
&VALUE = &MYFIELD.ShortTranslateValue;
If ALL(&VALUE) Then
   /* do processing */
End-if;
```

See Also

Chapter 12, Field Class , Value, page 869, Chapter 12, Field Class , LongTranslateValue, page 862.

ShowRequiredFieldCue

Description

With PeopleTools 8, an asterisk (*) is displayed on pages beside fields that are defined as Required in PeopleSoft Application Designer. You can use this property to specify whether this asterisk, also called the **required field cue**, is displayed for a particular field.

For example, many fields are made required or non-required either procedurally or through PeopleCode. This means they aren't defined as Required in PeopleSoft Application Designer, and the end-user may be confused. For these fields, you can use this property.

Note. This property affects only fields where a required field cue is otherwise permissible. That is, regardless of the setting of the property, no cue is ever shown on a pushbutton, a display-only field, and so on.

This property is read-write.

SqlText

Description

This property is valid only for fields that have a dynamic view as their prompt record. If you set SqlText to a non-null value, that text is used instead of the dynamic view's normal text used for prompting.

Suppose you wanted to have a different prompt table depending on the settings of other fields in the row. Normally you could use %EDITTABLE to dynamically specify the prompt table you want. However in this case there are too many possible combinations of values, which would require too many views. Furthermore, the values are customizable by the end-user or the application, which means even if you, the developer, wanted to, you couldn't provide all the combinations of views necessary. However you can generate the desired SQL text for the view in PeopleCode based on what the user enters.

If you use a dynamic view as the prompt table, and have the dynamic view contain a SQL object that is updated from PeopleCode, you could achieve this functionality. However, a SQL object is a shared object, so if multiple users used the same page, they overwrite each other's settings and the SQL object contains the SQL for the most recent user. Similarly if a single user had multiple rows on a page, the SQL object is valid only for the most recent row. This means if the user went to another row and did a prompt, they would get the wrong values again.

The purpose of this property is to enable you to specify the generated SQL text independently for each occurrence in each transaction. It enables you to override the text of a dynamic view being used as a prompt table on a field by field basis.

It is up to the developer to verify that the text specified for this property is valid, that is, that it selects the correct number of fields for the record definition, and so on.

This property is read-write.

StoredFormat

Description

This property returns the custom character format (as a string) for the field executing the property.

If the field doesn't have a custom format associated with it, the user receives a runtime error message.

If the field has a display format associated with it, you can change that using the DisplayFormat property.

This property is read-only.

Example

```
WinMessage("The character field's custom stored format is : " | ⇒
&CHARACTER.StoredFormat);

/* this assumes that &CHARACTER is a custom formatted */
/* field */
```

See Also

Chapter 12, Field Class , DisplayFormat, page 853.

Style

Description

If a page has a style sheet associated with it, this property can be used to specify a different style class with a field. This property overrides only the existing style. It does not change it. The next time the page is accessed, the original style is used.

This property takes a string value.

This property is read-write.

Example

The following example associates a test field first with a style class depending on the value of the field.

```
Local Field &field;

&field = GetField();

If TESTFIELD1 = 1 Then;
   &field.Style = "PSHYPERLINK";
End-If;

If TESTFIELD1 = 2 Then;
   &field.Style = "PSIMAGE";
End-If;
```

Field with PSHYPERLINK style

Field with PSIMAGE style

Type

Description

This property returns the type of field. The values can be one of the following strings:

- CHAR
- DATE
- DATETIME
- IMAGE (for static images)
- IMAGEREFERENCE
- LONGCHAR
- NUMBER
- SIGNEDNUMBER
- TIME

Note. Fields of type Attachment have a type of IMAGE.

This property is read-only.

Example

```
If &CHARACTER.Type = "NUMBER" Then
   /* perform processing */
Else
   /* error processing */
End-If;
```

Value

Description

This property contains the current value of the field, converted to an appropriate PeopleCode data type.

In most contexts, the Value property can be used to assign a new value to the field. However, there are some restrictions:

- You cannot assign the Value property in any RowSelect PeopleCode program.

- You cannot assign the current record field value to the Value property in any FieldEdit PeopleCode program.

Note. If you're accessing a field based on the translate table, Value returns only the one letter XLAT value. To get the full XLAT value, use the LongTranslateValue or ShortTranslateValue properties.

This property is read-write.

Considerations Using INSTALLATION or OPTIONS Tables

When using the INSTALLATION or OPTIONS table, you cannot use the **Value** property. You must use the old style format, not the field object format. For example, the following code is invalid:

```
If %Page = Page.GP_RUN_TYPE And
      INSTALLATION.TL.Value = "N" Then
   &RS2.HideAllRows();
```

While the following code is valid:

```
If %Page = Page.GP_RUN_TYPE And
      INSTALLATION.TL = "N" Then
   &RS2.HideAllRows();
```

Considerations Using Character Values

Previously, if you had an edit-box field, and if the end-user selected the value in it and deleted the value, leaving the field empty, the value of the field in PeopleCode was not an empty (zero-length) string.

Now, the value of such a field *is* an empty (zero-length) string.

In addition, if the user adds one or more space characters to a field, the field still returns a Null string.

The following is how to check for this:

```
If (fieldname.Value = "") Then
```

Example

```
&CHARACTER.Value = "Hello";
```

See Also

Chapter 12, Field Class , LongTranslateValue, page 862, Chapter 12, Field Class , ShortTranslateValue, page 866.

Visible

Description

This property is True if this field is visible in the page displaying it. Setting this property to False hides the field. Because every field is implicitly associated with a rowset, row, and record, setting the Visible property for a field on the first page of a component hides only that field. If that field is repeated on other pages in the component, the other occurrences of the field aren't hidden.

This property is read-write.

Example

The following code hides the field &ROUND_OPTION based on the value of AVG_DFLT_OPTION:

```
Local field &ROUND_OPTION;

&ROUND_OPTION = GetField();

If AVG_DFLT_OPTION = "Y" Then
   &ROUND_OPTION.SetDefault();
   &ROUND_OPTION.Visible = True;
Else
   &ROUND_OPTION.Visible = False;
End-If;
```

CHAPTER 13

File Class

This chapter describes the File class, which provides methods and properties for reading from and writing to external files and discusses the following topics:

- Data type of a file object
- Scope of a file object
- File security considerations
- File access interruption recovery
- Plain text files
- Automatic PeopleCode generation
- File layout error processing
- File class built-in functions
- File class methods
- File class properties
- File layout examples
- Multiple file layouts

See Also

PeopleSoft Integration Tools and Utilities, File Layouts and Data Interchange

Understanding File Layout

PeopleTools supports reading and writing to plain text files, and to files that have a format based on a File Layout definition that has been created in PeopleSoft Application Designer.

- If the file is a plain text file, data is read or written using text strings.
- If the file is based on a File Layout, you can use text strings, rowset, or record objects.

This simplifies reading, writing, and manipulating hierarchical transaction data with PeopleCode.

File layout methods and properties are noted as such in their descriptions.

See Also

Chapter 13, File Class , Plain Text Files , page 876

Chapter 13, File Class , File Layout Examples , page 901

PeopleSoft Integration Tools and Utilities, File Layouts and Data Interchange

Data Type of a File Object

The File Object is an instance of the File class. A file object is declared using the File data type. For example,

```
Local File &MYFILE;
```

This creates an object &MYFILE of the class File.

Scope of a File Object

A file object can only be instantiated from PeopleCode. This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

A file object is passed to and returned from PeopleCode functions and methods as a reference, so the file object is never copied; rather, alternate identifiers are used to refer to the same object. Any change made to a file using one identifier has the same effect as it would with any other identifier for that file.

In the following code example, two variables, &F1 and F2, are declared by using the data type File. The file is opened using the GetFile built-in function. Next, &F1 is assigned to &F2. This does *not* copy &F1 to &F2. Instead, &F1 and &F2 both refer to the same object. Therefore, in the last step when &F2 is closed, &F1 is closed too.

```
Local File &F1, &F2;

&F1 = GetFile("somefile.txt", "R");
&F2 = &F1;
&F2.Close();  /* Now &F1 is also closed. */
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetFile , page 243

PeopleCode Developer's Guide, Understanding Objects and Classes in PeopleCode, Assigning Objects

File Security Considerations

When you're using file objects in PeopleCode that might run on a server, you must be aware of some security concerns. The underlying system doesn't provide security checks on access to the files. Instead, the PeopleCode programs use whatever authority the server process has, and *not* that of the user. This means you must write your PeopleCode program to prevent the user from reading or writing files that they should not. In particular, be especially wary of opening files where any part of the filename is derived from user input.

File Access Interruption Recovery

You can use the GetPosition or SetPosition methods to minimize the loss of work in the event of a system failure during file access. To use them and recover from access interruptions, you must access the file in Update mode using the GetFile built-in function or the Open method, specifying the mode parameter **"U"**. You need to use this mode in anticipation of possible interruptions if you want to recover from them later.

To start reading or writing from the beginning of a file, use SetPosition to set a read/write position of **0**, immediately after you open the file in Update mode.

Warning! In Update mode, any write operation clears the file of all data that follows the position you set.

When reading from or writing to a file, use GetPosition periodically to determine your current byte position in the file. This establishes a checkpoint to which you can return after a failure. You must save the checkpoint value in a separate file or a database so you can retrieve it later. How often you save a checkpoint depends on your requirements: the less work you want to redo, the more frequent your checkpoints should be. You may also want to save information identifying the data you're reading or writing at the time.

After a failure interrupts your file access, reopen the file in Update mode. You can then retrieve the last checkpoint value you saved, and use SetPosition to apply that value. Your read/write position in the file is the position of the last checkpoint, and you can continue reading or writing from there.

Use the Update mode with GetPosition and SetPosition only for recovering from interruptions as described here, or for starting at the beginning of the file again.

Note. The effect of the Update mode is not well defined for Unicode files. Use the Update mode only on files stored with the ANSI character set.

See Also

Chapter 13, File Class , Open, page 882

Chapter 13, File Class , SetPosition, page 890

Chapter 13, File Class , GetPosition, page 882

Chapter 1, PeopleCode Built-in Functions, GetFile , page 243

Plain Text Files

To read and write from plain text files involves reading and writing strings of data. These text strings can be manipulated with built-in string functions, like RTrim, Find, Replace, and so on.

Note. If your data is hierarchical in nature, or based on existing PeopleSoft records or pages, use a File Layout definition for reading and writing your data, rather than doing it line by line (or field by field.)

The following example creates an array of array of string, then reads in two files, one into each "column" of the array. The **Names** file contains names; the **Numbers** file contains employee numbers. The **ReadLine** method reads each successive line in a file, until it reaches the end of the file.

Notice that the first file is opened using **GetFile**. The second file is not opened using **GetFile**, but rather with **Open**. After the data is read into the array, you can do processing on the data. The end of the program writes the changes back to the files, using the **WriteLine** method, which includes a system end of line character at the end of every line.

```
Local array of array of string &BOTH;
Local File &MYFILE;
Local string &HOLDER;

/* Create empty &BOTH array */
&BOTH = CreateArrayRept(CreateArrayRept("", 0), 0);

/* Read first file into first column */

&MYFILE = GetFile("names.txt", "R");
While &MYFILE.ReadLine(&HOLDER);
   &BOTH.Push(&HOLDER);
End-While;

/* read second file into second column */

&MYFILE.Open("numbers.txt", "R");
&LINENO = 1;
While &MYFILE.ReadLine(&HOLDER);
   If &LINENO > &BOTH.Len Then
      /* more number lines than names, use a null name */
      &BOTH.Push(CreateArray("", &HOLDER));
   Else
      &BOTH[&LINENO].Push(&HOLDER);
   End-If;
   &LINENO = &LINENO + 1;
End-While;

/* if more names than numbers, add null numbers */
For &LINENO = &LINENO to &BOTH.Len
      &BOTH[&LINENO].Push("");
End-For;
```

```
&MYFILE.Close();
/* do processing with array */
/* write data back to files */

&MYFILE1 = GetFile("names.txt", "A");
&MYFILE2 = GetFile("numbers.txt", "A");

/* loop through array and write to files */

For &I = 1 To &BOTH.Len
   &STRING1 = &BOTH[&I][1];
   &MYFILE1.writeline(&STRING1);
   &STRING2 = &BOTH[&I][2];
   &MYFILE2.writeline(&STRING2);
End-For;

&MYFILE1.Close();
&MYFILE2.Close();
```

See Also

Chapter 13, File Class , File Layout Examples , page 901

Automatic PeopleCode Generation

After you create a File Layout definition, you can use PeopleCode to access it. This PeopleCode can be long and complex. Rather than write it directly, you can drag and drop the File Layout definition from Application Designer Project View into an open PeopleCode edit pane. This is primarily used for importing data. Application Designer analyzes the definition and generates initial PeopleCode as a template, which you can modify to meet your requirements.

The following is just a snippet of the code that is generated:

```
Function EditRecord(&REC As Record) Returns boolean ;
   Local integer &E;
REM   &REC.ExecuteEdits(%Edit_Required + %Edit_DateRange + %Edit_YesNo + ⇒
%Edit_TranslateTable + %Edit_PromptTable + %Edit_OneZero);
   &REC.ExecuteEdits(%Edit_Required + %Edit_DateRange + %Edit_YesNo + ⇒
%Edit_OneZero);
   If &REC.IsEditError Then
      For &E = 1 To &REC.FieldCount
         &MYFIELD = &REC.GetField(&E);
         If &MYFIELD.EditError Then
            &MSGNUM = &MYFIELD.MessageNumber;
            &MSGSET = &MYFIELD.MessageSetNumber;
            &LOGFILE.WriteLine("****Record:" | &REC.Name | ", Field:" |⇒
 &MYFIELD.Name );
            &LOGFILE.WriteLine("****" | MsgGet(&MSGSET, &MSGNUM, ""));
```

```
         End-If;
      End-For;
      Return False;
   Else
      Return True;
   End-If;
End-Function;
.
.
.
```

See Also

PeopleSoft Integration Tools and Utilities, File Layouts and Data Interchange

File Layout Error Processing

If an error occurs on any field in any record of a rowset object populated with the ReadRowset method, the rowset object's property IsEditError returns True. For example, you can use the method ExecuteEdits on a record, to verify that the data in the record is valid (has the correct format, the right data type, and so on.) This type of error is indicated by the IsEditError.

In some instances, however, the rowset object won't receive the error; in that case the file object's IsError property returns True. To discover all field errors, check both properties after executing ReadRowset.

To determine which field has the error, you must examine the EditError property of every field in the rowset to find the one returning True. You can then examine that field's MessageSetNumber and MessageNumber properties to determine the relevant error message. The following example shows how this might be done:

```
&MYFILE.Open(&SOMENAME, "R");
&MYFILE.SetFileLayout(FILELAYOUT.SOMELAYOUT);
&MYROWSET = &MYFILE.ReadRowset();
If &MYFILE.IsError Then
  For &I = 1 to &MYROWSET.ActiveRowCount
    If &MYROWSET.GetRow(&I).IsEditError Then
      &ROW = &MYROWSET.GetRow(&I);
      For &J = 1 to &ROW.RecordCount
        If &ROW.GetRecord(&J).IsEditError Then
          &REC = &ROW.GetRecord(&J);
          For &K = 1 to &REC.FieldCount
            If &REC.GetField(&K).EditError Then
              /* Examine the field's
                 MessageSetNumber and MessageNumber properties,
                 and respond accordingly */
         End-If;
          End-For;
      End-If;
      End-For;
    End-If;
```

```
  End-For;
End-If;

&MYFILE.Close();
```

Note. Only field errors set the IsError, IsEditError, or EditError properties. All other errors triggered by File class methods terminate the PeopleCode program.

See Also

Chapter 13, File Class , IsError, page 899

Chapter 13, File Class , ReadRowset, page 886

Chapter 13, File Class , IgnoreInvalidId, page 899

Chapter 27, Rowset Class , IsEditError, page 1506

Chapter 25, Record Class , ExecuteEdits, page 1448

Chapter 17, Message Class , ExecuteEdits, page 987

Chapter 12, Field Class , MessageNumber, page 863

Chapter 12, Field Class , MessageSetNumber, page 863

File Class Built-in Functions

Chapter 1, PeopleCode Built-in Functions, FileExists , page 184

Chapter 1, PeopleCode Built-in Functions, FindFiles , page 189

Chapter 1, PeopleCode Built-in Functions, GetFile , page 243

File Class Methods

In this section, we discuss the File class methods. The methods are discussed in alphabetical order.

Close

Syntax

Close()

Description

The **Close** method discards any changes that haven't been written to the external file, disassociates the file object from the external file, releases all resources connected with the file object, and causes the file object to go out of scope.

You cannot use any methods or properties on the object after it is closed. You must get another file object (using **GetFile**) and instantiate another file object after you use Close.

Parameters

None.

Returns

None.

Example

```
&MYFILE.Open("somefile.txt", "W", %FilePath_Relative);
&MYFILE.WriteLine("Some text.");
&MYFILE.Close();
```

See Also

Chapter 13, File Class , Open, page 882, Chapter 13, File Class , IsOpen, page 900, Chapter 1, PeopleCode Built-in Functions, GetFile , page 243.

CreateRowset

Syntax

CreateRowset()

Description

The **CreateRowset** method is a file layout method. It instantiates a PeopleCode rowset object containing one unpopulated row, based on the file layout definition specified with SetFileLayout.

Note. You must specify a file layout using the SetFileLayout method before using **CreateRowset**, otherwise it returns NULL.

See Chapter 13, File Class , SetFileLayout, page 890.

After the empty rowset object has been created, you can use Rowset class methods such as Select or Fill, and built-in functions such as GetLevel0 or GetRowset, to populate the rowset with data. After the rowset contains data, you can use the method WriteRowset to write the data to a file.

Don't use **CreateRowset** when reading from a file. Instead, use the ReadRowset method to instantiate and populate rowsets with data from the file.

Parameters

None.

Returns

An empty rowset object.

Example

```
&SOMEROWSET = &MYFILE.CreateRowset();
```

The following rowset is created from a file object, then populated with data using the GetLevel0 function:

```
&FILEROWSET = &MYFILE.CreateRowset();
&FILEROWSET = GetLevel0();
&MYFILE.WriteRowset(&FILEROWSET, True);
```

The following rowset is created from a file object, then populated with data using Fill method:

```
&FILEROWSET = &MYFILE.CreateRowset();
&NUM_READ = &FILEROWSET.Fill("where MYRECORD = :1", &UVAL);
```

See Also

Chapter 13, File Class , ReadRowset, page 886, Chapter 13, File Class , SetFileLayout, page 890, Chapter 13, File Class , WriteRowset, page 896, Chapter 27, Rowset Class , page 1479.

Delete

Syntax

Delete()

Description

Use the **Delete** method to delete an *open* file. You cannot delete a closed file.

Parameters

None.

Returns

None.

Example

```
/* Open a temporary file. Use the "N" parameter to ensure
the file is new. */

&MyFile = GetFile("temp.txt", "N");

/* do processing using the temporary file */

/* delete the temporary file */

&MyFile.Delete();
```

See Also

Chapter 13, File Class , Open, page 882, Chapter 13, File Class , Close, page 879, Chapter 1, PeopleCode Built-in Functions, GetFile , page 243.

GetPosition

Syntax

GetPosition()

Description

The **GetPosition** method returns the current read or write position in the external file. GetPosition works only with a file that was opened in Update mode. This method is designed to be used in combination with the SetPosition method to establish checkpoints for restarting during file access.

Note. The effect of the Update mode is not well defined for Unicode files. Use the Update mode and GetPosition only on files stored with the ANSI character set.

Parameters

None.

Returns

A number representing the current read or write position in the file.

Note. When you use ReadRowset to read from a file in Update mode, the CurrentRecord property returns, the entire record just read. The current read/write position is at the *end* of the CurrentRecord, that is, just past the end of the rowset.

Example

The following example opens a file in Update mode, and saves the current position after each read operation:

```
&MYFILE.Open(&SOMENAME, "U");
If &MYFILE.IsOpen Then
   while &MYFILE.ReadLine(&SOMESTRING)
      &CURPOS = &MYFILE.GetPosition();
      /* Save the value of &CURPOS after each read,
         and process the contents of each &SOMESTRING */
   End-While;
End-If;
&MYFILE.Close();
```

See Also

Chapter 13, File Class , Open, page 882, Chapter 13, File Class , SetPosition, page 890, Chapter 13, File Class , File Access Interruption Recovery , page 875, Chapter 1, PeopleCode Built-in Functions, GetFile , page 243.

Open

Syntax

Open(*filespec*, *mode* [, *charset*] [, *pathtype*])

Description

The **Open** method associates the file object with an external file for input or output.

You can use the GetFile built-in function to access an external file, but each execution of GetFile instantiates a new file object. If you plan to access only one file at a time, you need only one file object. Use GetFile to instantiate a file object for the first external file you access, then use Open to associate the same file object with as many different external files as you want. You need to instantiate multiple file objects with GetFile only if you expect to have multiple files open at the same time.

If the file object is currently associated with a file, that file is first closed.

Working With Relative Paths

If you specify a relative path, that path is appended to the path specified by an environmental variable, if the variable exists, in a particular order. Possible environmental variables include PS_FILEDIR, PS_SERVDIR and TEMP.

If the first environmental variable in order is found, that is the one that's used. If it isn't found, the next one is searched for, and if found, is used, and so on.

The environmental variables that are used are different, depending on whether the machine where the code is running is an application server. If the code is running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]

If the code is *not* running on an application server, the following environmental variables are valid, and are searched for in the following order:

1. PS_FILEDIR[\relative_path]
2. PS_SERVDIR\Files[\relative_path]
3. TEMP[\relative_path]

The PS_FILEDIR environmental variable is *not* initialized and set to a path automatically. If you want to use this environmental variable, you must set it yourself.

Note. Your system security should verify if a user has the correct permissions before allowing access to a drive (for example, if a user changes their **TEMP** environmental variable to a network drive they don't normally have access to, your system security should detect this.)

Parameters

Parameter	Description
filespec	Specify the name, and optionally, the path, of the file you want to open.
mode	A string indicating the manner in which you want to access the file. The mode can be one of the following: **"R"** (Read mode): opens the file for reading, starting at the beginning. **"W"** (Write mode): opens the file for writing.

Warning! When you specify Write mode, any existing content in the file is discarded and is overwritten.

"A" (Append mode): opens the file for writing, starting at the end. Any existing content is retained.

"U" (Update mode): opens the file for reading or writing, starting at the end. Any existing content is retained. Use this mode and the GetPosition and SetPosition methods to maintain checkpoints of the current read/write position in the file.

In Update mode, any write operation clears the file of all data that follows the position you set.

Note. Currently, the effect of the Update mode and the GetPosition and SetPosition methods is not well defined for Unicode files. Use the Update mode only on files stored with the ANSI character set.

"E" (Conditional "exist" read mode): opens the file for reading only if it exists, starting at the beginning. If it doesn't exist, **Open** has no effect. Before attempting to read from the file, use the IsOpen property to confirm that it's open.

"N" (Conditional "new" write mode): opens the file for writing, only if it doesn't already exist. If a file by the same name already exists, **Open** has no effect. Before attempting to write to the file, use the IsOpen property to confirm that it's open. You can insert an asterisk (*****) in the filename to ensure that a new file is created. The system replaces the asterisk with numbers starting at 1 and incrementing by 1, and checks for the existence of a file by each resulting name in turn. It uses the first name for which a file *doesn't* exist. In this way you can generate a set of automatically numbered files. If you insert more than one asterisk, all but the first one are discarded.

Parameter	Description
charset	A string indicating the character set you expect when you read the file, or the character set you want to use when you write to the file. You can abbreviate ANSI to "A" and Unicode to "U". All other character sets must be spelled out in full, for example, ASCII, THAI, or UTF8. If you don't specify a character set, the default value is ANSI. You can also use a record field value to specify the character set (for example, RECORD.CHARSET.)

Note. If you attempt to read data from a file using a different character set than was used to write that data to the file, the methods used generate a runtime error or the data returned is unusable.

Parameter	Description
pathtype	If you have prepended a path to the file name, use this parameter to specify whether the path is an absolute or relative path. The valid values for this parameter are: • %FilePath_Relative (default) • %FilePath_Absolute If you don't specify *pathtype* the default is %FilePath_Relative. If the path is an absolute path, whatever path you specify is used verbatim. You must specify a drive letter and the complete path. You can't use any wildcards when specifying a path. The Component Processor automatically converts platform-specific separator characters to the appropriate form for where your PeopleCode program is executing. On a WIN32 system, UNIX "/" separators are converted to "\", and on a UNIX system, WIN32 "\" separators are converted to "/".

Note. The syntax of the file path does *not* depend on the file system of the platform where the file is actually stored; it depends only on the platform where your PeopleCode is executing.

Returns

None.

Example

The following example opens an existing Unicode file for reading:

```
&MYFILE.Open(&SOMENAME, "E", "U");
If &MYFILE.IsOpen Then
   while &MYFILE.ReadLine(&SOMESTRING)
      /* Process the contents of each &SOMESTRING */
   End-While;
   &MYFILE.Close();
End-If;
```

The following example opens a numbered file for writing in ANSI format, without overwriting any existing files:

```
&MYFILE.Open("C:\temp\item*.txt", "N", %FilePath_Absolute);
If &MYFILE.IsOpen Then
   &MYFILE.WriteLine("Some text.");
   &MYFILE.Close();
End-If;
```

See Also

Chapter 13, File Class , IsOpen, page 900, Chapter 13, File Class , SetPosition, page 890, Chapter 13, File Class , GetPosition, page 882, Chapter 13, File Class , Close, page 879, Chapter 13, File Class , File Access Interruption Recovery , page 875, Chapter 1, PeopleCode Built-in Functions, GetFile , page 243.

ReadLine

Syntax

ReadLine(*string*)

Description

The **ReadLine** method reads one line of text from the external file. The line includes the newline character, but ReadLine strips out the newline character and inserts the result into the string variable *string*.

When ReadLine is executed, it moves the starting point for the next read operation to the end of the text just retrieved, so each line in the file can be read in turn by subsequent ReadLine operations. When no more data remains to be read from the file, ReadLine returns False, and clears the string variable of any content.

Parameters

Parameter	Description
string	A string variable that receives the input text.

Returns

A Boolean value: True if the method succeeds, False otherwise. The return value is *not* optional, it is required.

Example

The following example reads a file called &MYFILE and puts each line as a separate element in an array.

```
Local File &MYFILE;
Local array of string &MYARRAY;
Local string &TEXT;

&MYFILE = GetFile("names.txt", "R");
&MYARRAY = CreateArrayRept("", 0);
While &MYFILE.ReadLine(&TEXT);
   &MYARRAY.Push(&TEXT);
End-While;
&MYFILE.Close();
```

See Also

Chapter 13, File Class , ReadRowset, page 886, Chapter 13, File Class , WriteString, page 897, Chapter 13, File Class , WriteLine, page 892.

ReadRowset

Syntax

```
ReadRowset()
```

Description

The **ReadRowset** method is a file layout method. It instantiates a PeopleCode rowset object based on the file layout definition, then populates the rowset with one transaction from the file. A transaction is considered to be one instance of level zero data contained in a file record, plus all of its subordinate data. If you put more than one segment at level zero, each segment is read in sequence.

Note. You must specify a file layout using the SetFileLayout method before using **ReadRowset**, otherwise it will return NULL.

When ReadRowset is executed, it moves the starting point for the next read operation to the beginning of the next rowset, so each transaction in the file can be read in turn by subsequent ReadRowset operations. When no more data remains to be read from the file, ReadRowset returns NULL.

If you're using the SetFileId method with ReadRowset to process an input file based on multiple file layouts, ReadRowset returns NULL when it reads a FileId file record (line) between the rowsets.

When ReadRowset returns NULL, you can use the IsFileId property to determine if you've reached the end of the file or a FileId record.

If the ReadRowset encounters a line in the file containing the FileId, and the lines following this are *not* a new rowset, the process considers it to be an invalid FileId. You can specify whether to ignore the invalid record or terminate the PeopleCode with the IgnoreInvalidId property.

Note. If you're using the SetFileId method with ReadRowset to process an input file based on multiple layouts, FileId file records between the rowsets are considered to be valid file records, and won't generate any errors, regardless of the state of the IgnoreInvalidId property.

See Chapter 13, File Class , SetFileLayout, page 890.

Considerations for Using Dates With ReadRowset

Single digits in dates in the form MMDDYY or MMDDYYYY must be padded with zeros. That is, if the date in your data is February 3, 2000, the form must be:

```
02/03/2000
```

or

```
02/03/00
```

The following is *not* valid.

```
2/3/00
```

Considerations Using XML With ReadRowset

If all of the fields in the file layout are not present in the XML, no data is written to the database. If there are additional tags in the XML, they are ignored.

Parameters

None.

Returns

A populated rowset.

Example

The following example reads and processes an entire file. The data in the file is based on a single File layout definition:

```
&MYFILE.GetFile(&SOMENAME, "R");
If &MYFILE.SetFileLayout(FILELAYOUT.SOMELAYOUT) then
   &SOMEROWSET = &MYFILE.ReadRowset();
```

```
   While &SOMEROWSET <> NULL
      /* Process the contents of each &SOMEROWSET */
      &SOMEROWSET = &MYFILE.ReadRowset();
   End-While;
End-If;
&MYFILE.Close();
```

See Also

Chapter 13, File Class , CurrentRecord, page 898, Chapter 13, File Class , IgnoreInvalidId, page 899, Chapter 13, File Class , ReadLine, page 885, Chapter 13, File Class , SetFileLayout, page 890, Chapter 13, File Class , WriteRecord, page 894, Chapter 13, File Class , WriteRowset, page 896, Chapter 13, File Class , Multiple File Layouts , page 915, Chapter 13, File Class , ReadRowset Example , page 910, Chapter 27, Rowset Class , IsEditError, page 1506.

SetFileId

Syntax

SetFileId(*fileid*, *position*)

Description

The **SetFileId** method is a file layout method. If your input file contains data based on more than one File Layout definition, you must use this method in combination with the CurrentRecord and IsNewFileId properties and the ReadRowset method to process the file correctly.

Note. SetFileId, CurrentRecord and IsNewFileId don't apply to CSV and XML format input files. You can use only FIXED format files to implement multiple file layouts.

See Chapter 13, File Class , CurrentRecord, page 898; Chapter 13, File Class , IsNewFileId, page 900 and Chapter 13, File Class , ReadRowset, page 886.

At each point in the input file where the structure of the rowset changes, there must be an extra line in the file containing the file record (line), the **FileId** file record (line), that signifies that the change. Each FileId file record must have the following components:

- A value that designates it as a Fileid. Only one FileId value is necessary throughout the file, such as "999".
- A name field that identifies the file layout needed for the rowset that follows. This field could contain the name of the file layout as it's defined in PeopleSoft Application Designer.

These lines containing the FileId are *not* part of any rowset. They can contain other information, which will be disregarded by the system. The FileId identifier and the file layout names aren't automatically stored anywhere; they exist only in the input file's FileId file records and in the PeopleCode.

To process an input file that requires multiple file layouts:

1. Use SetFileLayout to specify the file layout definition to use.

   If you're specifying the initial file layout for this file, it doesn't have to be the correct one. You can determine the correct file layout to use for the first rowset during a subsequent step.

2. Use SetFileId to specify the value of the FileId field, *fileid*, that's used throughout the file to designate FileId file records.

Note. After each execution of SetFileLayout, the SetFileId setting is disabled. Be sure to re-specify the FileId value if you expect the file layout to change, so the system continues looking for the next FileId file record.

3. Use ReadRowset to read the next rowset from the file.

   The current file record is also stored in the CurrentRecord property as a string. The PeopleCode process determines whether it's the beginning of a new rowset, or a FileId file record according to the *fileid* you specified. If it's a FileId file record, the process sets the IsNewFileId property to True; if not, it sets the IsNewFileId property to False.

   Note. If this is the first execution of ReadRowset on the file, it returns NULL upon encountering the initial FileId file record, but still stores that file record in CurrentRecord and sets IsNewFileId accordingly.

4. If the rowset returned isn't NULL, process that rowset as necessary.
5. If IsNewFileId is False, go back to step 3 to continue reading rowsets from the file. If IsNewFileId is True, examine CurrentRecord to determine which new file layout to use, and go back to step 1.

You can repeat this procedure one rowset at a time, proceeding through the remainder of the input file. When ReadRowset returns NULL, no more rowset data is available.

Parameters

Parameter	Description
fileid	Specify the value of the FileId field used in the current file.
position	Specify the column in the FileId file record where the FileId field starts. The default is 1.

Returns

None.

Example

```
&rsFile = &MYFILE.ReadRowset();
&CurrentRecord = &MYFILE.CurrentRecord;
&IsNewFileID = &MYFILE.IsNewFileId;
If &MYFILE.IsNewFileId Then
   &FILELAYOUT = FindFileID(&CurrentRecord);
   &MYFILE.SetFileLayout(@("FileLayout." | &FILELAYOUT));
   &MYFILE.SetFileId("999", 1);
End-If;
```

See Also

Chapter 13, File Class , CurrentRecord, page 898, Chapter 13, File Class , IsNewFileId, page 900, Chapter 13, File Class , ReadRowset, page 886, Chapter 13, File Class , SetFileLayout, page 890, Chapter 13, File Class , File Layout Examples , page 901, *PeopleSoft Integration Tools and Utilities*, File Layouts and Data Interchange.

SetFileLayout

Syntax

SetFileLayout(FILELAYOUT.*filelayoutname*)

Description

The **SetFileLayout** method is a file layout method. It associates a specific file layout definition with the file object executing this method, providing easy access to rowset data. The file object must be associated with an open file. With file layout definitions, you can read and write rowsets as easily as strings. If the file object isn't open or the definition doesn't exist, SetFileLayout fails.

You must execute SetFileLayout before you can use any other file layout methods or properties with a file object, otherwise those methods and properties return NULL or False.

Parameters

Parameter	Description
filelayoutname	Specify as a string the name of an existing file layout definition created in PeopleSoft Application Designer.

Returns

A Boolean value: True if the method succeeds, False otherwise.

Example

The following example opens a data file, associates a file layout definition with it, reads and processes the first rowset from it, and closes the file.

```
&MYFILE.Open(&SOMENAME, "E");
If &MYFILE.SetFileLayout(FILELAYOUT.SOMELAYOUT) then
   &SOMEROWSET = &MYFILE.ReadRowset();
   /* Process the contents of &SOMEROWSET */
Else
   /* Error - SetFileLayout failed */
End-If;
&MYFILE.Close();
```

See Also

Chapter 13, File Class , CurrentRecord, page 898, Chapter 13, File Class , CreateRowset, page 880, Chapter 13, File Class , ReadRowset, page 886, Chapter 13, File Class , IgnoreInvalidId, page 899, Chapter 13, File Class , WriteRowset, page 896, Chapter 13, File Class , File Layout Examples , page 901, *PeopleSoft Integration Tools and Utilities*, File Layouts and Data Interchange.

SetPosition

Syntax

SetPosition(*position*)

Description

The **SetPosition** method sets the current read or write position in the external file associated with the file object executing this method. SetPosition works only with a file, which is opened in Update mode, and is designed to be used in combination with the GetPosition method to recover from interruptions during file access.

To start reading or writing from the beginning of a file, you must use SetPosition to set a read/write position of **0**, immediately after you open the file in Update mode.

Parameters

Parameter	Description
position	The byte position in the file at which you want to continue reading or writing. This should either be **0** or a value you previously obtained with the GetPosition method and saved in a separate file or a database.

Warning! In Update mode, any write operation clears the file of all data that follows the position you set.

Note. Use **SetPosition** only for recovering from file access interruptions. Supplying your own value for SetPosition isn't recommended, except for position 0.

Returns

None.

Example

The following example reopens a file in Update mode, and sets the read position to the last saved checkpoint:

```
&MYFILE = GetFile(&SOMENAME, "U");
If &MYFILE.IsOpen Then
   /* Retrieve the value of the last saved checkpoint, &LASTPOS */
   &MYFILE.SetPosition(&LASTPOS);
   while &MYFILE.ReadLine(&SOMESTRING)
      /* Process the contents of each &SOMESTRING */
   End-While;
   &MYFILE.Close();
End-If;
```

Note. The effect of the Update mode is not well defined for Unicode files. Use the Update mode and **SetPosition** only on files stored with the ANSI character set.

See Also

Chapter 13, File Class , GetPosition, page 882, Chapter 13, File Class , Open, page 882, Chapter 13, File Class , File Access Interruption Recovery , page 875, Chapter 1, PeopleCode Built-in Functions, GetFile , page 243.

SetRecTerminator

Syntax

SetRecTerminator(*Terminator*)

Description

The **SetRecTerminator** method is a file layout method.

Use this method to specify the string of characters that indicate the end of a file record. Read operations use the value of SetRecTerminator to determine where each file record ends and the next one starts, and write operations append the value of SetRecTerminator to each record written.

This value defaults to the newline character appropriate to the platform where the file is being stored:

- a linefeed on UNIX systems
- a carriage return/linefeed combination on Windows systems

You need to specify a different record terminators only if you anticipate that part of the data in a file field that includes the newline character used on the storage platform; you must assign the record terminator a unique string that you know does not appear in a file field.

If you set the record terminator to Null ("") you eliminate linefeeds or carriage returns.

Parameters

Parameter	Description
Terminator	Specify the value you want used for the record terminator.

Returns

None.

Example

```
&File = GetFile(......);

if &File.IsOpen then
   /* set the terminator to be NULL */
   Local String &Term = "";
   &File.SetRecTerminator(&Term);
```

WriteLine

Syntax

WriteLine(*string*)

Description

The **WriteLine** method writes one string of text, *string*, to the output file associated with the file object executing this method, followed by a newline character appropriate to the platform where the file is being written. To build a single line using multiple strings, use the WriteString method.

Parameters

Parameter	Description
string	The string of text to be written.

Returns

None.

Example

The following example adds a line of text to an existing file:

```
&MYFILE.Open("somefile.txt", "A");
&MYFILE.WriteLine("This is the last line in the file.");
&MYFILE.Close();
```

The following example converts a file where the fields are separated with tabs into a file where the fields are separated with commas.

```
Local File &TABFILE, &CSVFILE;
Local string &FILE_NAME, &DATA, &NEWDATA;

&FILE_NAME = "Test.txt";
&TAB = Char(9);

&TABFILE = GetFile(&FILE_NAME, "r");
&FileName = &TABFILE.Name;
&POS = Find(".", &FileName);
&NEWFILE_NAME = Substring(&FileName, 1, &POS) | "dat";
&CSVFILE = GetFile(&NEWFILE_NAME, "N", %FilePath_Absolute);
If &TABFILE.IsOpen And
      &CSVFILE.IsOpen Then
   While &TABFILE.ReadLine(&DATA);
      &NEWDATA = Substitute(&DATA, &TAB, ",");
      &CSVFILE.WriteLine(&NEWDATA);
   End-While;
   &TABFILE.Close();
   &CSVFILE.Close();
End-If;
```

See Also

Chapter 13, File Class , ReadLine, page 885, Chapter 13, File Class , WriteString, page 897, Chapter 13, File Class , WriteRaw, page 893.

WriteRaw

Syntax

WriteRaw(*RawBinary*)

Description

The **WriteRaw** method writes the contents of *RawBinary* to a file. This can be used for writing images, messages, or other types of raw binary data to a file.

Parameters

Parameter	Description
RawBinary	Specify the raw binary to be written to the file.

Returns

None.

Example

The following example writes employee photos (GIF files) from a record to a file.

```
Local File &FILE;
Local Record &REC;
Local SQL &SQL;

&REC = CreateRecord(Record.EMPL_PHOTO);
&SQL = CreateSQL("%SelectAll(:1)", Record.EMPL_PHOTO);

&FILE = GetFile("C:\temp\EMPL_PHOTO.GIF", "w", "a", %FilePath_Absolute);

While &SQL1.Fetch(&REC)
   &FILE.WriteRaw(&REC.EMPLOYEE_PHOTO.Value);
End-While;

&FILE.Close();
```

See Also

Chapter 13, File Class , WriteLine, page 892, Chapter 13, File Class , WriteString, page 897.

WriteRecord

Syntax

WriteRecord(*record*)

Description

The **WriteRecord** method is a file layout method. It writes the contents of the record object *record*, to the output file. (Remember, a record object contains only one row of data from an SQL table.)

You can use this method to build a transaction in the output file, one file record at a time, without having to instantiate and populate a rowset object. You can apply this method to any record whose structure and name matches that of a record defined in the current file layout.

Note. You must execute the SetFileId method from the file object before using WriteRecord, otherwise it returns False.

See Chapter 13, File Class , SetFileLayout, page 890.

Note. When you're writing text to an XML file, special HTML characters, such as ampersands, lesser than or greater than symbols, and so on, are automatically converted to valid XML character string, such as &amp.

Considerations Using XML With File Definition Tags

File Definition Tags have no effect when importing data. However, generally, during export, the File Definition Tag is added at the start and end of the data to create a valid XML file.

The one exception is when the file to which the data is being written during export has been opened in append mode. At that time, the file definition tag is not taken into consideration.

Parameters

Parameter	Description
record	Specify the name of an existing record object to be written. You can use Rowset class methods such as GetField and built-in functions such as GetRecord to populate the record with data before writing it to the file.

Returns

A Boolean value: True if the method succeeds, False otherwise. This value is optional.

Example

The following example appends all the data in a record to an existing file:

```
Local File &MYFILE;

&MYFILE = GetFile("record.txt", "A");

If &MYFILE.IsOpen Then
   If &MYFILE.SetFileLayout(FILELAYOUT.VOL_TEST) Then
      &LN = CreateRecord(RECORD.VOLNTER_ORG_TBL);
      &SQL2 = CreateSQL("%Selectall(:1)", &LN);
      While &SQL2.Fetch(&LN)
         &MYFILE.WriteRecord(&LN);
      End-While;
   Else
      /* do error processing − filelayout not correct */
   End-If;
Else
   /* do error processing − file not open */
End-If;

&MYFILE.Close();
```

See Also

Chapter 13, File Class , CreateRowset, page 880, Chapter 13, File Class , ReadRowset, page 886, Chapter 13, File Class , SetFileLayout, page 890, Chapter 13, File Class , WriteRowset, page 896, Chapter 25, Record Class , page 1441, *PeopleCode Developer's Guide*, Data Buffer Access.

WriteRowset

Syntax

WriteRowset(*rowset* [, *Write_Data*])

Description

The **WriteRowset** method is a file layout method. It writes the contents of a rowset object, *rowset*, to the output file associated with the file object executing this method. Regardless of whether the rowset contains just one or more than one transaction (level 0 row), executing this method once writes the entire contents of the rowset to the output file.

Note. You must execute the SetFileLayout method from the file object before using WriteRowset, otherwise it returns False.

See Chapter 13, File Class , SetFileLayout, page 890.

WriteRowset writes a rowset to a file only if the data in the component buffer has changed. You must specify the *WriteData* parameter as True if you want the WriteRowset method to force the rowset to be written to the file even if the buffer has not been changed.

Note. When you're writing text to an XML file, special HTML characters, such as ampersands, lesser than or greater than symbols, and so on, are automatically converted to valid XML character string, such as &amp.

Considerations Using Fixed Length Files With Numeric Fields

All numeric fields are right-justified in when writing to fixed length files. In addition, zeros are padded to the right after the decimal point if required.

Considerations Using XML With File Definition Tags

File Definition Tags have no effect when importing data. However, generally, during export, the File Definition Tag is added at the starting and end of the data to create a valid XML file.

The one exception is when the file to which the data is being written during export has been opened in append mode. At that time, the file definition tag is not taken into consideration.

Parameters

Parameter	Description
rowset	Specify the name of an existing rowset object that was instantiated with the File class CreateRowset or ReadRowset method. You can use Rowset class methods such as Select and built-in functions such as GetLevel0 to populate the rowset with data before writing it to the file.
WriteData	Specify whether to write the rowset data to the file whether or not the data in the buffer has changed. This parameter takes a Boolean value: True, write the data to the buffer regardless, False, only write the data if changed. The default value for this parameter is False.

Returns

A Boolean value: True if the method succeeds, False otherwise.

Example

```
Local File &MYFILE;
Local Rowset &FILEROWSET;

&MYFILE = GetFile("c:\temp\EMP_CKLS.txt", "A", %FilePath_Absolute);
If &MYFILE.IsOpen Then
   If &MYFILE.SetFileLayout(FILELAYOUT.EMPL_CHECKLIST) Then
      &FILEROWSET = &MYFILE.CreateRowset();
      &FILEROWSET = GetLevel0();
      &MYFILE.WriteRowset(&FILEROWSET, True);
   Else
      /* file layout not found, do error processing */
   End-If;
Else
   /* file not opened, do error processing */
End-if;

&MYFILE.Close();
```

See Also

Chapter 13, File Class , CreateRowset, page 880, Chapter 13, File Class , ReadRowset, page 886, Chapter 13, File Class , SetFileLayout, page 890, Chapter 13, File Class , WriteRecord, page 894, Chapter 25, Record Class , page 1441, *PeopleCode Developer's Guide*, Data Buffer Access.

WriteString

Syntax

WriteString(*string*)

Description

The **WriteString** method writes one string of text to the output file associated with the file object executing this method, without any newline character. Each string written extends the current line in the file.

You can start a new line by using the WriteLine method to write the last part of the current line. WriteLine always adds a newline character appropriate to the platform where the file is being written, whether you supply a character string of any length, or a null string.

Parameters

Parameter	Description
string	A string variable containing the text to be written.

Returns

None.

Example

The following example opens an empty file, writes two lines of text to it without a final newline, and closes it:

```
&MYFILE.Open("somefile.txt", "W");
&MYFILE.WriteString("This is the first ");
&MYFILE.WriteString("line in the file.");
&MYFILE.WriteLine("");
&MYFILE.WriteString("This second line is not terminated.");
&MYFILE.Close();
```

See Also

Chapter 13, File Class , ReadLine, page 885, Chapter 13, File Class , WriteLine, page 892, Chapter 13, File Class , WriteRaw, page 893.

File Class Properties

In this section, we discuss each File Class property.

CurrentRecord

Description

This property is a file layout property. This property returns the current record as a string. **CurrentRecord** is used in combination with the SetFileId and ReadRowset methods and IsNewFileId property when reading files that contain data based on multiple file layouts.

If ReadRowset returns NULL, either the current record is a FileId record or the end of file has been reached. The IsNewFileId enables you to determine which. If it is a FileId, you can parse the string returned by CurrentRecord to determine the file layout definition to be used.

Note. SetFileId, CurrentRecord, and IsNewFileId don’t apply to CSV and XML format input files. You can implement multiple file layouts only with FIXED format files.

This property is read-only.

Example

```
&RECSTRING = &MYFILE.CurrentRecord;
```

See Also

Chapter 13, File Class , SetFileId, page 888, Chapter 13, File Class , ReadRowset, page 886, Chapter 13, File Class , IsNewFileId, page 900, Chapter 13, File Class , Multiple File Layouts , page 915.

IgnoreInvalidId

Description

This property is a file layout property that's used in combination with the ReadRowset method. It returns a Boolean value that specifies whether file records with invalid FileIds are ignored. Each time ReadRowset is executed, it may encounter a file record that doesn't qualify as part of the rowset because its File ID isn't part of the current file layout, or because it occurs in an invalid position in the rowset. If IgnoreInvalidId is False, the PeopleCode program terminates; if IgnoreInvalidId is True, ReadRowset ignores the invalid file record. The default value is True.

This property is read-write.

Example

```
&MYFILE.IgnoreInvalidId = False;
```

See Also

Chapter 13, File Class , ReadRowset, page 886.

IsError

Description

This property is a file layout property. It returns a Boolean value indicating whether a field error condition was generated by the last file layout method executed. If an error condition was generated, **IsError** returns True; if not, **IsError** returns False. The default value is False.

This property is read-only.

Example

The following example shows where IsError would be used:

```
&MYFILE.Open(&SOMENAME, "R");
&MYFILE.SetFileLayout(FILELAYOUT.SOMELAYOUT);
&MYROWSET = &MYFILE.ReadRowset();
If &MYFILE.IsError Then
   /* Examine the EditError property of each field in the rowset
      to find the one with the error, and respond accordingly */
End-If;
&MYFILE.Close();
```

See Also

Chapter 13, File Class , Multiple File Layouts , page 915.

IsNewFileId

Description

This property is a file layout property. It returns a Boolean value indicating whether a FileId file record has been encountered. When the ReadRowset method reads a transaction from an input file, it examines the current file record following the transaction. If that file record is a FileId file record, IsNewFileId returns True; if not, IsNewFileId returns False.

IsNewFileId is used in combination with the SetFileId method and CurrentRecord property when reading files that require multiple file layouts.

Note. SetFileId, CurrentRecord, and IsNewFileId don't apply to CSV and XML format input files. You can use only FIXED format files to implement multiple file layouts.

This property is read-only.

Example

```
&MYFILE.SetFileLayout(FILELAYOUT.SOMELAYOUT) then  /* Set the first layout */
&MYFILE.SetFileId("999", 1);  /* Set the FileId */
&SOMEROWSET = &MYFILE.ReadRowset();  /* Read the first rowset */
While &SOMEROWSET <> NULL
   If &MYFILE.IsNewFileId Then
      /* Examine &MYFILE.CurrentRecord for the next layout */
      /* Set the next layout */
      &MYFILE.SetFileId("999", 1);  /* Set the FileId */
   End-If;
   /* Process the current &SOMEROWSET */
   &SOMEROWSET = &MYFILE.ReadRowset();  /* Read the next rowset */
End-While;
```

See Also

Chapter 13, File Class , ReadRowset, page 886, Chapter 13, File Class , SetFileId, page 888, Chapter 13, File Class , CurrentRecord, page 898.

IsOpen

Description

This property returns a Boolean value indicating whether the file is open. If the file object is open, IsOpen returns True; if not, IsOpen returns False.

This property is read-only.

Example

The following example opens a file, writes a line to it, and closes it:

```
&MYFILE.Open("item.txt", "W");
If &MYFILE.IsOpen Then
   &MYFILE.WriteLine("Some text.");
```

```
   &MYFILE.Close();
End-If;
```

Name

Description

This property returns as a string the name of the external file. If no file is currently associated with the file object, **Name** returns a NULL string.

This property is read-only.

Example

```
&TMP = &MYFILE.Name;
```

File Layout Examples

If your data is hierarchical in nature, or based on existing PeopleSoft records or pages, you want to use a File Layout definition for reading and writing your data, rather than doing it line by line (or field by field.)

For example, suppose you wanted to write all the information from a record to a file. You can use the **WriteRecord** method to write all the data from the record, instead of having to loop through every field, find the value, and write it to the file.

In addition, you could write all the information from a transaction (or several transactions) from a page to a file. Each transaction can be considered a *rowset*. A rowset can contain more than one record and is generally composed in a hierarchical structure. You could create a File Layout definition that has the same structure as the page (or component), and use the **WriteRowset** method. If you have a file that contains data in the correct format, you can use the **ReadRowset** method to read the data from the file to the page.

Each file layout is associated with a *format*. This format specifies the type of data in the files. You specify the format as part of the File Layout Properties. You can only specify one format for a file layout. Available formats are:

- FIXED (default)
- CSV
- XML

The file layout methods and properties use this information to handle each file type in a transparent manner. Generally, you don't need to do anything different based on file type. Any exceptions are noted in the documentation.

Note. Unlike other PeopleTools definitions, records and field are *copied* to a File Layout definition. There are no pointers. This means if you change a record definition (add or remove a field) you must change the File Layout definition also. The changes are not automatically propagated. This is why the documentation refers to these elements as file records, file fields, and so on, to show that they are no longer part of the original definition they were created from.

See Using Standalone Rowsets for more examples of writing from and reading to files using File Layout and standalone rowsets.

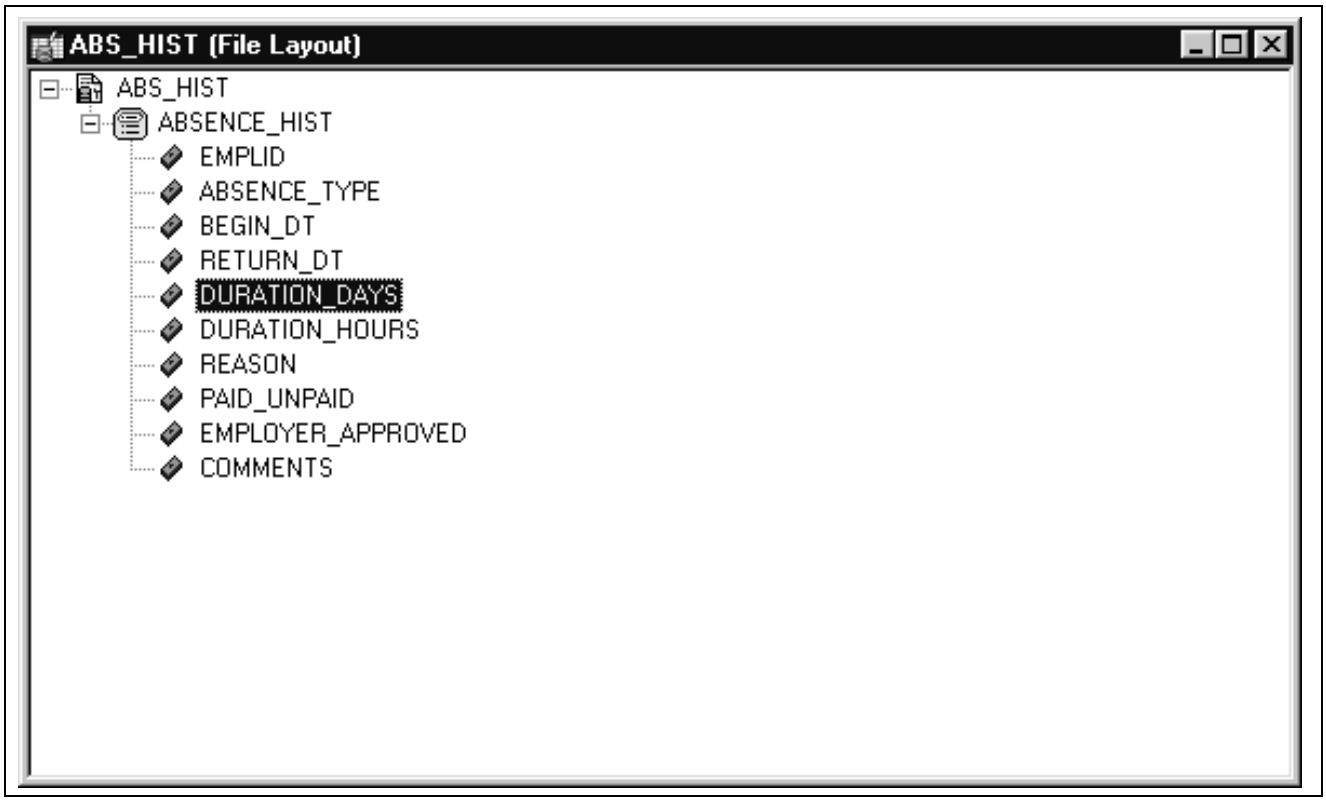
See *PeopleSoft Integration Tools and Utilities*, File Layouts and Data Interchange.

See *PeopleCode Developer's Guide*, Data Buffer Access.

See *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using Standalone Rowsets.

WriteRecord Example

In the following example, the File Layout definition is based on the record ABSENCE_HISTORY, and looks like this:



Example File Layout definition (ABS_HIST)

You should note the following about the using **WriteRecord**:

- Not all the fields in the File Layout definition and the record have to match. The WriteRecord method, like all File Layout methods, applies only to the *like-named* fields. If there are additional fields in the record or in the File Layout definition, they are ignored.
- The WriteRecord method writes only to *like-named* records. If you rename a record after you use it to create a File Layout definition, you must rename it to the exact same name in your File Layout. Because WriteRecord writes like-named records, the same file layout definition can contain more than one record.
- The WriteRecord method takes a *record object* as its parameter. A populated record object references a *single row* of data in the SQL table. This is why a SQL Fetch statement is used in a condition around the WriteRecord method. This fetches every row of data from the SQL table, then writes it to the file.

The following code writes the ABSENCE_HIST record to the file record.txt.

```
Local Record &RecLine;
Local File &MYFILE;
Local SQL &SQL2;

&MYFILE = GetFile("record.txt", "A");

If &MYFILE.IsOpen Then
   If &MYFILE.SetFileLayout(FileLayout.ABS_HIST) Then
      &RecLine = CreateRecord(RECORD.ABSENCE_HIST);
      &SQL2 = CreateSQL("%Selectall(:1)", &RecLine);
      While &SQL2.Fetch(&RecLine)
         &MYFILE.WriteRecord(&RecLine);
      End-While;
   Else
      /* do error processing − filelayout not correct */
   End-If;
Else
   /* do error processing − file not open */
End-If;

&MYFILE.Close();
```

If you wanted to write only changed records to the file, you could add the following code that is set in bold font:

```
While &SQL2.Fetch(&RecLine)
   If &RecLine.IsChanged Then
         &MYFILE.WriteRecord(&RecLine);
   End-If;
End-While;
```

The following is the first part of a sample data file created by the previous code. The field format is FIXED:

```
8001       VAC 09/12/1981 09/26/1981 14  0                                P Y
8001       VAC 03/02/1983 03/07/1983 5   0                                P Y
8001       VAC 08/26/1983 09/10/1983 13  0                                P Y
8105       CNF 02/02/1995 ??/??/     0   0                                U N
8516       MAT 06/06/1986 08/01/1986 56  0                                P Y
8516       SCK 08/06/1988 08/07/1988 1   0                                P Y
8516       VAC 07/14/1987 07/28/1987 14  0                                P Y
8553       JUR 12/12/1990 12/17/1990 5   0   Local Jury Duty              P N
8553       MAT 02/20/1992 10/01/1992 224 0   Maternity Leave              U N
8553       MAT 08/19/1994 03/01/1995 194 0   Maternity-2nd child          U Y
8553       PER 04/15/1993 04/19/1993 4   0                                U N⇒
 Personal Day required
8553       SCK 01/28/1987 01/30/1987 2   0   Hong Kong Flu                P N
8553       SCK 08/02/1988 08/03/1988 1   0   Sick                         P N
8553       SCK 09/12/1995 09/13/1995 1   0                                P N
8641       VAC 06/01/1988 06/15/1988 14  0                                P Y
8641       VAC 07/01/1989 07/15/1989 14  0                                P Y
G001       MAT 07/02/1991 09/28/1991 88  0   3-month Maternity leave       P Y⇒
 Maternity will be paid as 80% of Claudia’s current salary.
```

If a record in the File Layout definition has a File Record ID, each line in the file referencing this record is prefaced with this number. The File Record ID is not a field in the data itself. It is also referred to as the *rowid*. File Record IDs can be useful when the file you're producing refers to more than one record.

File Record IDs can be used only with File Layout definitions that have a type of FIXED. If a File Layout definition has only one level then File Record IDs can be omitted. But for File Layout definitions with more than one level, you *must* use File Record IDs.

The following is sample file, produced with the same code, but with a File Record ID of 101 added:

```
101 8001       VAC 09/12/1981 09/26/1981 14  0                                P Y   ⇒

101 8001       VAC 03/02/1983 03/07/1983 5   0                                P Y   ⇒

101 8001       VAC 08/26/1983 09/10/1983 13  0                                P Y   ⇒

101 8105       CNF 02/02/1995 ??/??/     0   0                                U N   ⇒

101 8516       MAT 06/06/1986 08/01/1986 56  0                                P Y   ⇒

101 8516       SCK 08/06/1988 08/07/1988 1   0                                P Y   ⇒

101 8516       VAC 07/14/1987 07/28/1987 14  0                                P Y   ⇒

101 8553       JUR 12/12/1990 12/17/1990 5   0   Local Jury Duty              P N   ⇒

101 8553       MAT 02/20/1992 10/01/1992 224 0   Maternity Leave              U N   ⇒

101 8553       MAT 08/19/1994 03/01/1995 194 0   Maternity-2nd child          U Y   ⇒

101 8553       PER 04/15/1993 04/19/1993 4   0                                U N⇒
 Personal Day required
101 8553       SCK 01/28/1987 01/30/1987 2   0   Hong Kong Flu                P N   ⇒

101 8553       SCK 08/02/1988 08/03/1988 1   0   Sick                         P N   ⇒

101 8553       SCK 09/12/1995 09/13/1995 1   0                                P N   ⇒

101 8641       VAC 06/01/1988 06/15/1988 14  0                                P Y   ⇒

101 8641       VAC 07/01/1989 07/15/1989 14  0                                P Y   ⇒

101 G001       MAT 07/02/1991 09/28/1991 88  0   3-month Maternity leave       P Y⇒
 Maternity will be paid as 80% of Claudia's current salary.
```

Note. File positions start at 1, not 0. When you specify a File Record ID, make sure that the starting position of the Record ID is greater than 0.

ReadRecord Example

This following example uses the same File Layout definition as the previous example. The file format is CSV. The fields are separated by commas and delimited by double-quotes.

```
"8001","VAC","09/12/1981","09/26/1981","14","0","","P","Y",""                    ⇒

"8001","VAC","03/02/1983","03/07/1983","5","0","","P","Y",""                     ⇒

"8001","VAC","08/26/1983","09/10/1983","13","0","","P","Y",""                    ⇒

"8105","CNF","02/02/1995","??/??/","0","0","","U","N",""                         ⇒

"8516","MAT","06/06/1986","08/01/1986","56","0","","P","Y",""                    ⇒

"8516","SCK","08/06/1988","08/07/1988","1","0","","P","Y",""                     ⇒

"8516","VAC","07/14/1987","07/28/1987","14","0","","P","Y",""                    ⇒

"8553","JUR","12/12/1990","12/17/1990","5","0","Local Jury Duty","P","N",""      ⇒

"8553","MAT","02/20/1992","10/01/1992","224","0","Maternity Leave","U","N",""    ⇒

"8553","MAT","08/19/1994","03/01/1995","194","0","Maternity-2nd child","U","Y",""⇒

"8553","PER","04/15/1993","04/19/1993","4","0","","U","N","Personal Day required"⇒

"8553","SCK","01/28/1987","01/30/1987","2","0","Hong Kong Flu","P","N",""        ⇒

"8553","SCK","08/02/1988","08/03/1988","1","0","Sick","P","N",""                 ⇒

"8553","SCK","09/12/1995","09/13/1995","1","0","","P","N",""                     ⇒

"8641","VAC","06/01/1988","06/15/1988","14","0","","P","Y",""                    ⇒

"8641","VAC","07/01/1989","07/15/1989","14","0","","P","Y",""                    ⇒

"G001","MAT","07/02/1991","09/28/1991","88","0","3-month Maternity leave","P","Y","⇒
Maternity will be paid as 80% of Claudia’s current salary."
```

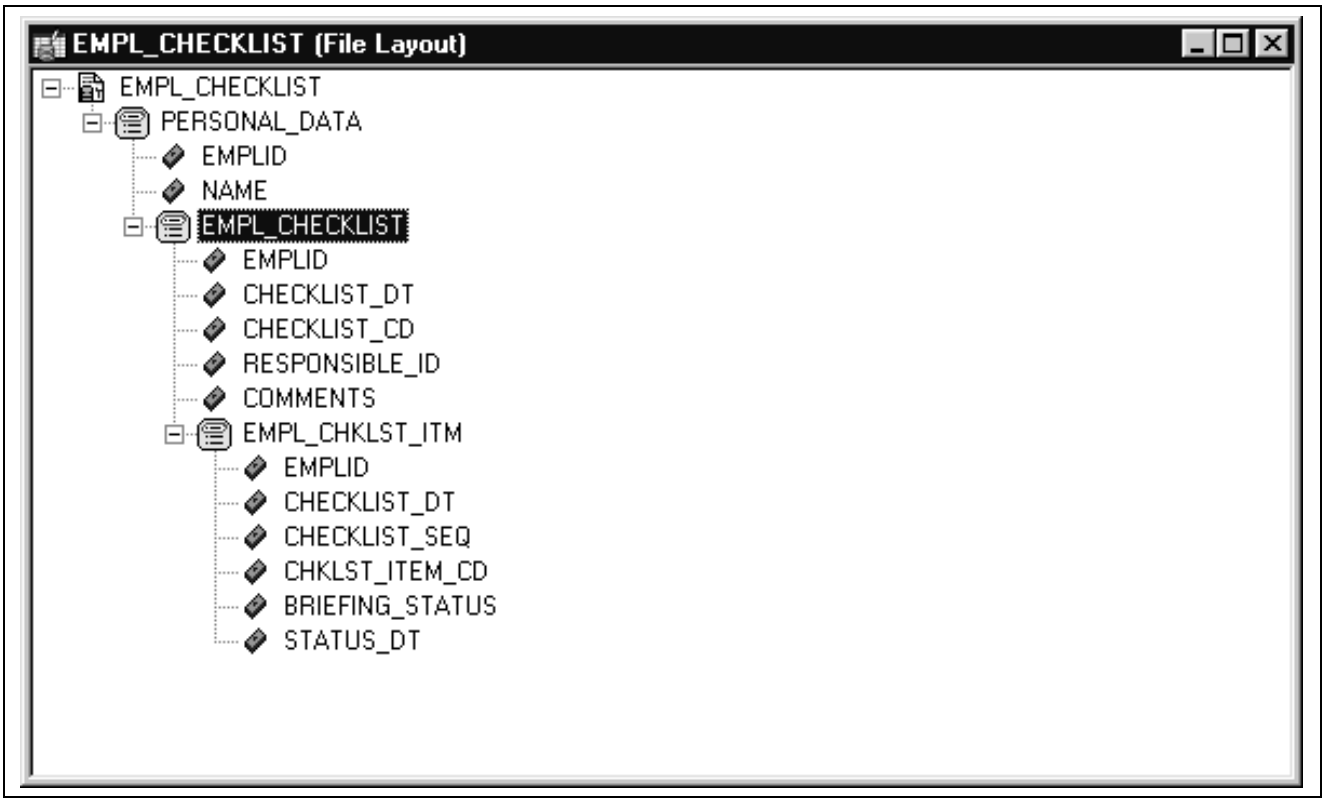
To read in the previous CSV file we use the following PeopleCode. It reads the file into a temporary record. First each line of the file is read into a string. The string is split into an array, with the value of each field in the array becoming an element in the array. The value of each field in the record is assigned a value from the array. After additional processing (for example, converting strings into dates or numbers, verifying data, and so on) the record can be inserted into the database. To insert the final data into the database, this code must be associated with a PeopleCode event that allows database updates, that is, SavePreChange, WorkFlow, SavePostChange, and so on. This code could also be used as part of an Application Engine program.

```
Local File &MYFILE;
Local Record &REC;
Local array of string &ARRAY;
```

```
&MYFILE = GetFile("c:\temp\vendor.txt", "R", %FilePath_Absolute);
&REC = CreateRecord(RECORD.ABS_HIST_TEST);
&ARRAY = CreateArrayRept("", 0);

If &MYFILE.IsOpen Then
   If &MYFILE.SetFileLayout(FILELAYOUT.ABS_HIST) Then
      While &MYFILE.ReadLine(&STRING);
         &ARRAY = Split(&STRING, ",");
         For &I = 1 To &REC.FieldCount
            &REC.GetField(&I).Value = &ARRAY[&I];
         End-For;
      /* do additional processing here for converting values */
         &REC.Insert();
      End-While;
   Else
      /* do error processing – filelayout not correct */
   End-If;
Else
   /* do error processing – file not open */
End-If;

&MYFILE.Close();
```

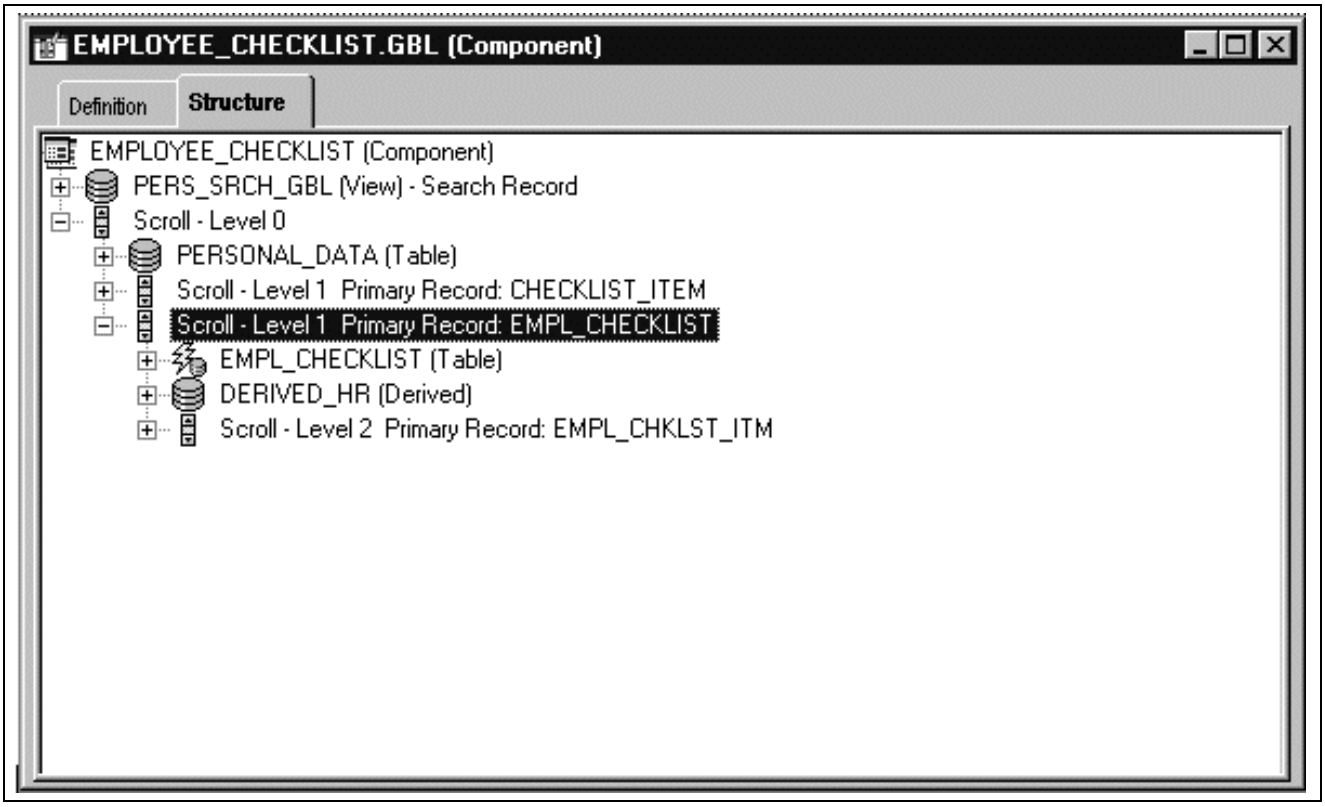
WriteRowset Example

In the following example, the File Layout definition is based on the component EMPL_CHECKLIST, and looks like this:



Example File Layout definition (EMPL_CHECKLIST)

Here's the structure of the component EMPLOYEE_CHECKLIST:



EMPLOYEE_CHECKLIST Component structure

Note that:

- Every *field* in the two structures don't have to match (that is, every field or record that's in the file layout doesn't have to be in the component, and vice versa.)
- The two *structures* must be the same. That is, if the component has PERSONAL_DATA at Level 0, and EMPL_CHECKLIST at Level 1, the file layout must have the same hierarchy.

The following example uses the previous File Layout definition to copy data from the EMPL_CHECKLIST page into a file.

The **CreateRowset** function creates an empty rowset that has the structure of the file layout definition. The **GetRowset** function is used to get all the data from the component and copy it into the rowset. The **GetLevel0** function copies all *like-named* fields to *like-named* records. The **WriteRowset** method writes all the component data to the file. Because this code runs on the server, an absolute file path is used.

See Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268.

Example

The following is the PeopleCode for this example.

```
Local File &MYFILE;
Local Rowset &FILEROWSET;

&MYFILE = GetFile("c:\temp\EMP_CKLS.txt", "A", %FilePath_Absolute);
If &MYFILE.IsOpen Then
   If &MYFILE.SetFileLayout(FILELAYOUT.EMPL_CHECKLIST) Then
      &FILEROWSET = &MYFILE.CreateRowset();
      &FILEROWSET = GetLevel0();
      &MYFILE.WriteRowset(&FILEROWSET, True);
   Else
      /* file layout not found, do error processing */
   End-If;
Else
   /* file not opened, do error processing */
End-if;

&MYFILE.Close();
```

The following is a sample data file created by the previous code:

```
8113       Frumman,Wolfgang
           08/06/1999 000001           8219        Going to London office
                     100     000015 I 08/06/1999
                     200     000030 I 08/06/1999
                     300     000009 I 08/06/1999
                     400     000001 I 08/06/1999
                     500     000011 I 08/06/1999
                     600     000002 I 08/06/1999
                     700     000021 I 08/06/1999
                     800     000024 I 08/06/1999
                     900     000004 I 08/06/1999
```

```
                    1000    000006 I 08/06/1999
            09/06/1999 000004           7707        What to do after he arrives
                    100     000022 I 08/06/1999
                    200     000008 I 08/06/1999
                    300     000018 I 08/06/1999
                    400     000019 I 08/06/1999
8101        Penrose,Steven
            07/06/1999 000006           8229        New hire
                    1       000033 I 08/06/1999
                    2       000034 I 08/06/1999
                    3       000035 I 08/06/1999
                    4       000036 I 08/06/1999
                    5       000037 I 08/06/1999
                    6       000038 I 08/06/1999
                    7       000039 I 08/06/1999
                    8       000040 I 08/06/1999
                    9       000041 I 08/06/1999
                    10      000042 I 08/06/1999
```

When you create the File Layout definition, you can choose for each field whether to inherit a value from a higher level record field. If a value is inherited, it is written only *once* to the file, the first time it's encountered.

In the previous data example, there are three records: PERSONAL_DATA, EMPL_CHECKLIST, and EMPL_CHKLST_ITM. The field EMPLID is on all three records, but is ritten to the file only the first time a new value is encountered. So, the value for EMPLID is inherited by EMPL_CHECKLIST, and the value for EMPL_CHKLST_ITM is inherited from PERSONAL_DATA. The field CHECKLIST_DT is on EMPL_CHECKLIST and EMPL_CHKLST_ITM. However, the field value appears only once, in the parent record, and not in the child record.

If a record in the File Layout definition has a File Record ID, each line in the file referencing this record will be prefaced with this number. The File Record ID is not a field in the data itself. It is also referred to as the *rowid*. File Record IDs can be useful when the file being created refers to more than one record. File Record IDs can be used only with File Layout definitions that have a type of FIXED.

The following is sample file, produced with the same code, but with a File Record IDs added to all the records. 001 was added to the first level, 002 to the second, and 003 to the third.

```
001 8113        Frumman,Wolfgang                                                ⇒

002             08/06/1999      000001 8219         Going to London office      ⇒

003                         100    000015 I 10/13/1999                          ⇒

003                         200    000030 I 10/13/1999                          ⇒

003                         300    000009 I 10/13/1999                          ⇒

003                         400    000001 I 10/13/1999                          ⇒

003                         500    000011 I 10/13/1999                          ⇒

003                         600    000002 I 10/13/1999                          ⇒
```

```
003                      700    000021 I 10/13/1999                          ⇒
003                      800    000024 I 10/13/1999                          ⇒
003                      900    000004 I 10/13/1999                          ⇒
003                      1000   000006 I 10/13/1999                          ⇒
002            09/06/1999      000004 7707        What to do after he arrives ⇒
003                      100    000022 I 10/13/1999                          ⇒
003                      200    000008 I 10/13/1999                          ⇒
003                      300    000018 I 10/13/1999                          ⇒
003                      400    000019 I 10/13/1999                          ⇒
001 8101       Penrose,Steven                                                 ⇒
002            10/13/1999      000006 8229        New hire                    ⇒
003                      1      000033 I 10/13/1999                          ⇒
003                      2      000034 I 10/13/1999                          ⇒
003                      3      000035 I 10/13/1999                          ⇒
003                      4      000036 I 10/13/1999                          ⇒
003                      5      000037 I 10/13/1999                          ⇒
003                      6      000038 I 10/13/1999                          ⇒
003                      7      000039 I 10/13/1999                          ⇒
003                      8      000040 I 10/13/1999                          ⇒
003                      9      000041 I 10/13/1999                          ⇒
003                      10     000042 I 10/13/1999
```

ReadRowset Example

The following program reads all the rowsets from a file and updates a work scroll with that data. The work scroll isn't hidden on the page: it's created in the Component buffer from existing records using CreateRowset. The structure of the work scroll and the file layout are identical: that is, they're composed of two records, and the names of the records in the file layout are exactly the same as the names of the record definitions.

```
Local File &CHARTINPUT_F;
Local Rowset &INPUT_ROWSET, &TEMP_RS, &WORK_DATA;
Local Record &WRK_DATA;

&filename = "c:\temp\test.txt";
If FileExists(&filename, %FilePath_Absolute) Then
   &CHARTINPUT_F = GetFile(&filename, "R", "A", %FilePath_Absolute);
Else
   Exit;
End-If;

&CHARTINPUT_F.SetFileLayout(FileLayout.CHART_INFO);

/* Create rowset to be read into
NOTE that you have to start at LOWEST level of rowset */

&TEMP_RS = CreateRowset(RECORD.CHART_ITEM);
&WORK_DATA = CreateRowset(RECORD.CHART_DATA, &TEMP_RS);
&INPUT_ROWSET = CreateRowset(RECORD.CHART, &WORK_DATA);

While &INPUT_ROWSET <> Null
   &INPUT_ROWSET = &CHARTINPUT_F.ReadRowset();
   &INPUT_ROWSET.CopyTo(&WORK_DATA);
   /* do processing -- Though file may contain more than one level 0
   Component processor only allows one level 0 at a time */
End-While;
```

File Rowset Considerations

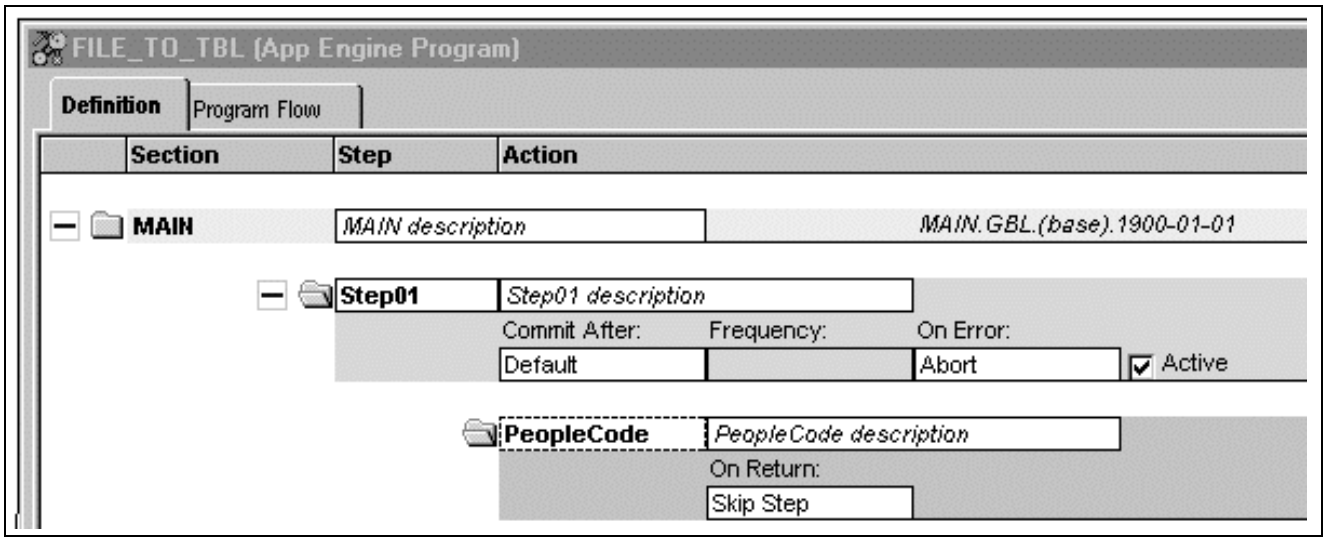
The following are considerations for when you use rowsets with files.

- Although you can create a File Layout definition with more than four levels of hierarchy, a rowset created from Component buffer data can contain only four levels (level 0 through 3). Any additional levels of data are ignored.
- **ReadRowset** populates the rowset with *one* transaction from the file. (A transaction is considered to be one instance of level zero data contained in a file record, plus all of its subordinate data.) **WriteRowset** writes one transaction to a file.

Application Engine Example

You can also use PeopleCode in an Application Engine program to either write to or read from files. This example isn't a proper Application Engine program: it contains the PeopleCode only for opening and reading from a file. However, it's included here as starting point for your own Application Engine programs.

Here is the Application Engine program:



Application Engine example program

Here is the PeopleCode in the step 1.

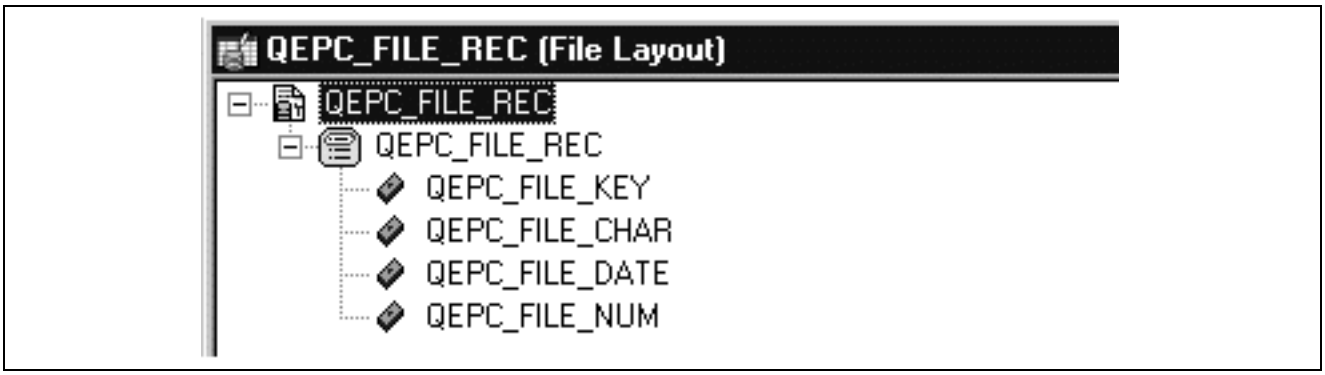
```
Local File &FILE;
Local Record &REC;
Local Rowset &FRS;

&FILE = GetFile("TEST.txt", "R");
&REC = CreateRecord(Record.QEPC_FILE_REC);
&SQL = CreateSQL("%Insert(:1)");

If Not &FILE.IsOpen Then
   Error ("TEST: failed file open");
Else
   If Not &FILE.SetFileLayout(FileLayout.QEPC_FILE_REC) Then
      Error ("TEST: failed SetFilelayout");
   Else
      &FRS = &FILE.ReadRowset();
      While &FRS <> Null
         &FRS.GetRow(1).QEPC_FILE_REC.CopyFieldsTo(&REC);
         &SQL.execute(&REC);
         &FRS = &FILE.ReadRowset();
      End-While;
   End-If;
   &FILE.Close();
End-If;
```

The example Application Engine program reads the following CSV file:

```
"TEST2","1ST","01/01/1901",10
"TEST2","2ND","01/01/1902",20
"TEST2","3RD","01/01/1903",30
"TEST2","4TH","01/01/1904",40
```

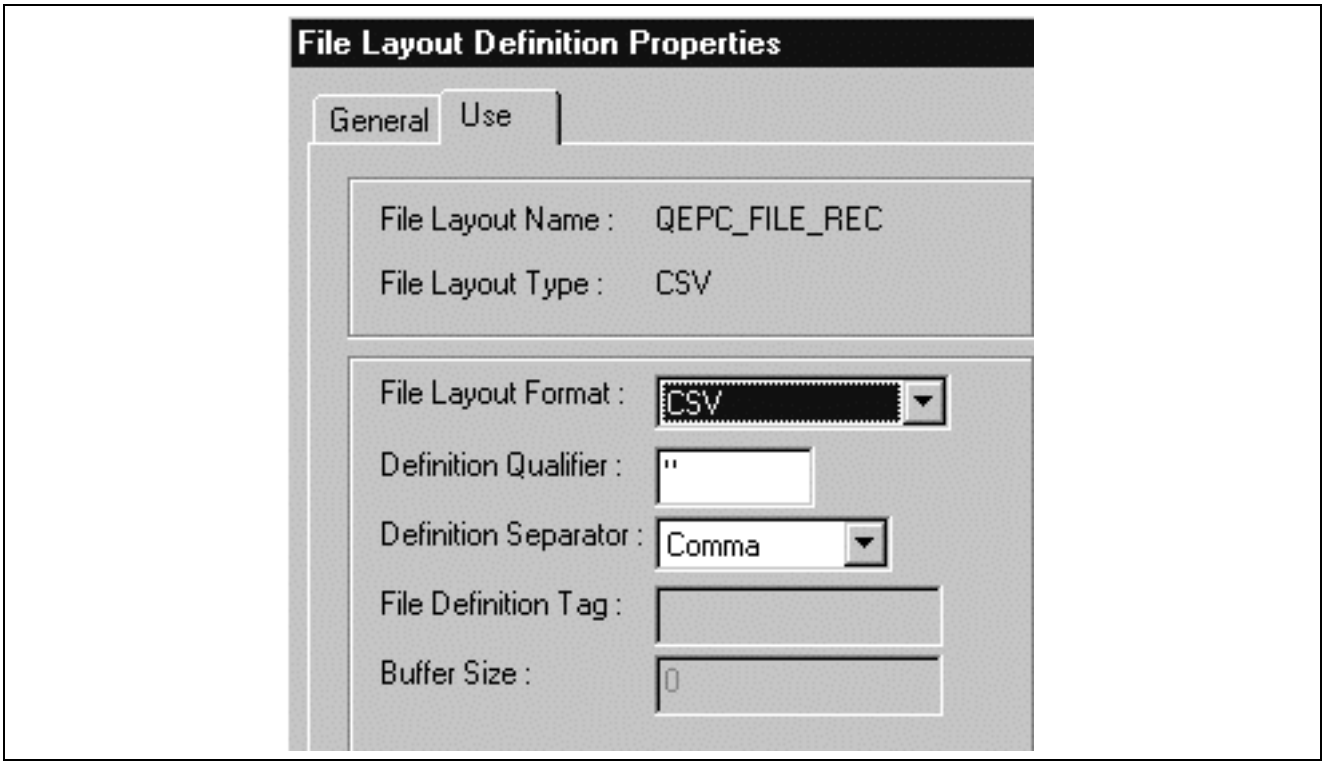
Note that the last field has no qualifier.

The File Layout used to read this record has the following form:



QEPC_FILE_REC File Layout

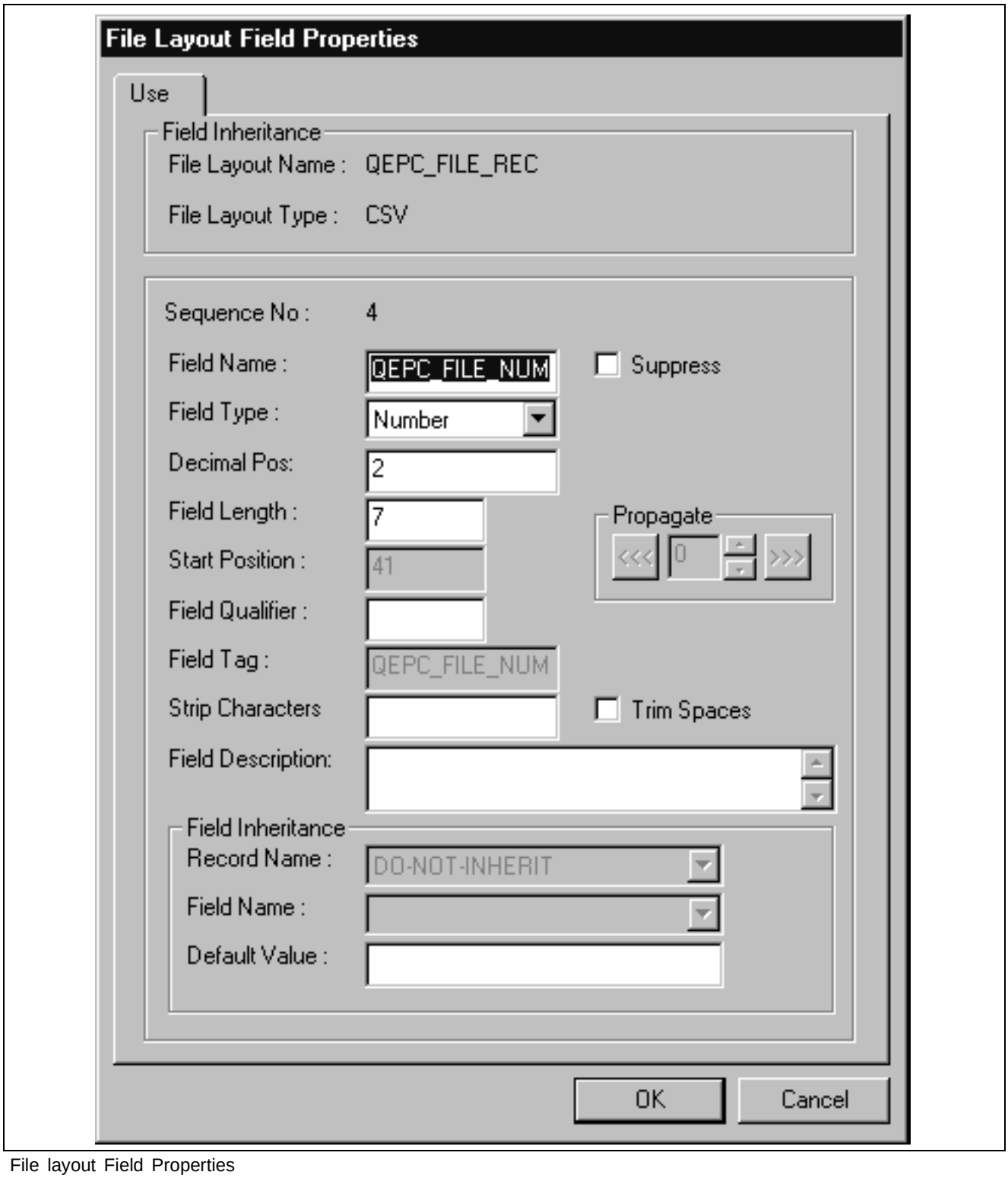
The properties for the QEPC_FILE_REC File Layout are as follows:



File Layout Definition Properties

Note that the Definition Qualifier is double-quotes ("), while the separator is a comma.

Remember, the last field didn't have a qualifier. To account for that, the field properties for this field must be edited, and a blank must be put in the **Field Qualifier** property.



File layout Field Properties

See *PeopleSoft Application Engine*, Introducing Application Engine.

Multiple File Layouts

In the previous examples, the input file contained rowsets based on a single File Layout definition. However, PeopleTools provides the functionality to process input files containing rowsets that require several *different* File Layouts.

Note. You can use only FIXED format files to implement multiple file layouts.

See the SetFileId method for details about handling multiple file layouts.

In this section, we discuss how to:

- Read multiple file layouts.
- Write multiple file layouts.

See Chapter 13, File Class , SetFileId, page 888.

Reading Multiple File Layouts

If your input file contains data based on more than one File Layout, it must contain an indicator, called a **FileId** that specifies:

- When a different File Layout definition should be used.
- Which File Layout definition should be used.

The FileId must be specified on a separate line and must precede *every* rowset that requires a layout different from the previous rowset. It isn't considered part of the rowset.

In the following example, the file contains two FileId lines; they use a file record ID that distinguishes them from the rowset data—in this case, "999".

```
999 PRODUCT  /* The following rowset uses the PRODUCT layout */
001  /* Level 0 record data */
101  /* Level 1 record data */
201  /* Level 2 record data */
201  /* Level 2 record data */
999 ORDER  /* The following two rowsets use the ORDER layout */
001  /* Level 0 record data */
111  /* Level 1 record data */
111  /* Level 1 record data */
001  /* Level 0 record data */
111  /* Level 1 record data */
111  /* Level 1 record data */
```

The FileId can contain any information you want that indicates which file layout to use; the "PRODUCT" and "ORDER" fields shown are just examples.

To read this file, you should do the following in your program:

1. Use the SetFileId method to specify the file record ID value.
2. Use the ReadRowset method to read the data.

3. Check if the rowset is NULL.

   NULL indicates you've reached either the end of the file or a new rowset.

4. Use the IsNewFileId property to check if the next line is a **FileId** file record (line).

- If IsNewFileld is False, you've reached the end of the file.
- If IsNewFileld is True, use the **CurrentRecord** property to determine which File Layout to use next.

The following example reads rowsets from a file. When it finds a new rowset (indicated by &IsNewFileld returning True) the value of &CurrentRecord is passed to a function that reads and evaluates the line, then returns the name of the new file layout. (The code for the function FindFileId is included at the start of the example.)

```
Local File &MYFILE;
Local Rowset &rsFile;
Local Record &rSomeRec1, &rSomeRec2;
Local SQL &SQL1;

Function FindFileID(&CurrentRecord As string) Returns string ;
   Evaluate RTrim(Substring(&CurrentRecord, 5, 50))
   When "SOME_REC1"
      &FILELAYOUT = "SOME_REC1";
   When "SOME_REC2"
      &FILELAYOUT = "SOME_REC2";
   End-Evaluate;
   Return &FILELAYOUT;
End-Function;

&rSomeRec1 = CreateRecord(Record.SOME_REC1);
&rSomeRec2 = CreateRecord(Record.SOME_REC2);
&SQL1 = CreateSQL("%Insert(:1)");
&MYFILE = GetFile("c:\temp\MULTI_FILE.out", "R", %FilePath_Absolute);

rem Set temporary first file layout;
&MYFILE.SetFileLayout(FileLayout.SOME_REC1);
&MYFILE.SetFileId("999", 1);

rem Read rowset to find actual first file ID;
&rsFile = &MYFILE.ReadRowset();
&CurrentRecord = &MYFILE.CurrentRecord;
&IsNewFileID = &MYFILE.IsNewFileId;
If &MYFILE.IsNewFileId Then
   &FILELAYOUT = FindFileID(&CurrentRecord);
   &MYFILE.SetFileLayout(@("FileLayout." | &FILELAYOUT));
   &MYFILE.SetFileId("999", 1);
End-If;

rem Read first 'real' rowset;
&rsFile = &MYFILE.ReadRowset();
&CurrentRecord = &MYFILE.CurrentRecord;
&IsNewFileID = &MYFILE.IsNewFileId;
```

```
While &rsFile <> Null Or
      &IsNewFileID
   If &MYFILE.IsNewFileId Then
      &FILELAYOUT = FindFileID(&CurrentRecord);
      &MYFILE.SetFileLayout(@("FileLayout." | &FILELAYOUT));
      &MYFILE.SetFileId("999", 1);
      If &IsNewFileID Then
         &rsFile = &MYFILE.ReadRowset();
         &CurrentRecord = &MYFILE.CurrentRecord;
         &IsNewFileID = &MYFILE.IsNewFileId;
      End-If;
   End-If;

   Evaluate &FILELAYOUT

   When "SOME_REC1"
      &rsFile(1).SOME_REC1.CopyFieldsTo(&rSomeRec1);
      &rSomeRec1.ExecuteEdits(%Edit_Required);
      If Not &rSomeRec1.IsEditError Then
         &SQL1.Execute(&rSomeRec1);
      End-If;
      Break;
   When "SOME_REC2"
      &rsFile(1).SOME_REC2.CopyFieldsTo(&rSomeRec2);
      &rSomeRec2.ExecuteEdits(%Edit_Required);
      If Not &rSomeRec2.IsEditError Then
         &SQL1.Execute(&rSomeRec2);
      End-If;
   End-Evaluate;

   &rsFile = &MYFILE.ReadRowset();
   &CurrentRecord = &MYFILE.CurrentRecord;
   &IsNewFileID = &MYFILE.IsNewFileId;
End-While;

&MYFILE.Close();
```

See Chapter 13, File Class , SetFileId, page 888; Chapter 13, File Class , ReadRowset, page 886 and Chapter 13, File Class , IsNewFileId, page 900.

Writing Multiple File Layouts

If you're writing files that contain data based on more than one File Layout definition, consider the following points:

- If the file is going to a third-party vendor, you should work with the third-party to determine what their requirements are for specifying the different data formats.
- If the file is going to be used by another PeopleSoft system, you must add the **FileId** between each rowset that requires a different layout. FileId file records are *not* part of any rowset. They

should be designed so they won't be mistaken for part of a rowset. You can create and write them to the file in many ways. The following are suggestions:

- Build each line as a string, using any of the built-in string manipulation functions, then write them to the file using the File class WriteLine or WriteString methods.
- Design a file layout consisting of a single file record definition for the FileId file records, then build the records using Record Class methods and functions, and write them to the file using the WriteRecord method.

The following code example writes each record from the level 1 scroll on a page to the file using a different File Layout. Between each WriteRowset the File ID file record is written to the file, describing the new File Layout being used.

```
Local File &MYFILE;
Local Rowset &FILEROWSET;
Local Record &REC1, &REC2;
Local SQL &SQL;

&MYFILE = GetFile("c:\temp\Records.txt", "W", %FilePath_Absolute);
If &MYFILE.IsOpen Then
   If &MYFILE.SetFileLayout(FileLayout.TREE_LEVEL) Then
      &REC1 = CreateRecord(Record.PSTREELEVEL);
      &FILEROWSET = &MYFILE.CreateRowset();
      &SQL = CreateSQL("%Selectall(:1)", &REC1);
      /* write first File ID to file */
      &MYFILE.WriteLine("999 FILE LAYOUT 1");
      While &SQL.Fetch(&REC1)
         &REC1.CopyFieldsTo(&FILEROWSET.GetRow(1).PSTREELEVEL);
         &MYFILE.WriteRowset(&FILEROWSET, True);
      End-While;
   Else
      /* file layout not found, do error processing */
   End-If;

   If &MYFILE.SetFileLayout(FileLayout.TREE_USERLEVEL) Then
      &REC2 = CreateRecord(Record.TREE_LEVEL_TBL);
      &FILEROWSET = &MYFILE.CreateRowset();
      &SQL = CreateSQL("%Selectall(:1)", &REC2);
      /* write second File ID to file */
      &MYFILE.WriteLine("999 FILE LAYOUT 2");
      While &SQL.Fetch(&REC2)
         &REC2.CopyFieldsTo(&FILEROWSET.GetRow(1).TREE_LEVEL_TBL);
         &MYFILE.WriteRowset(&FILEROWSET);
      End-While;
   Else
      /* file layout not found, do error processing */
   End-If;
Else
   /* file not opened, do error processing */
End-If;
&MYFILE.Close();
```

See Chapter 13, File Class , WriteLine, page 892; Chapter 13, File Class , WriteString, page 897 and Chapter 13, File Class , WriteRecord, page 894.

See Chapter 25, Record Class , page 1441.

CHAPTER 14

Grid Classes

This chapter provides an overview of Grid class and the GridColumn subobject and discusses the following topics:

- Shortcut considerations
- The Grid class in PeopleCode
- Grid or Grid Column declaration
- Scope of a Grid or Grid Column class
- Grid Class built-in function
- Grid Class reference
- Grid Column class

Understanding Grid and GridColumn Classes

A grid looks and behaves like a spreadsheet embedded in a page: it has column headings, row headings, cells, and horizontal and vertical scroll bars. It can be used instead of a single-level scroll. It is analogous to a scroll region on a page. Each row in a grid corresponds to a set of controls in a scroll occurrence. Each cell in a grid corresponds to a field on a page.

The Grid class enables you to instantiate only GridColumn objects, which in turn enables you to change grid column attributes without having to write PeopleCode that loops through every row of the grid.

Note. PeopleSoft builds a page grid one row at a time. Because the Grid class applies to a complete grid, you can't attach PeopleCode that uses the Grid class to events that occur before the grid is built; the earliest event you can use is the page Activate Event.

See Also

Chapter 14, Grid Classes , GridColumn Class , page 926

PeopleSoft Application Designer, Using Page Controls, Using

PeopleCode Developer's Guide, PeopleCode and the Component Processor, Activate Event

Shortcut Considerations

An expression of the form

```
&MYGRID.columnname.property
```

is converted to an object expression by using **GetColumn** *(columnname*). The following examples are equivalent.

Using *columnname:*

```
&MYGRIDCOLUMN = &MYGRID.CHECKLIST_ITEMCODE;
```

Using the GetColumn method:

```
&MYGRIDCOLUMN = &MYGRID.GetColumn("CHECKLIST_ITEMCODE");
```

The following examples are equivalent.

Using *columnname*:

```
&MYGRID.CHECKLIST_ITEMCODE.Visible = False;
```

Using the GetColumn method:

```
&MYGRID.GetColumn("CHECKLIST_ITEMCODE").Visible = False;
```

The Grid Class in PeopleCode

The PeopleCode grid object is a reference to a page runtime object for the grid. These particular page runtime objects aren't present until the Component is started.

Note. PeopleSoft builds a page grid one row at a time. Because the Grid class applies to a complete grid, you can't attach PeopleCode that uses the Grid class to events that occur before the grid is built; the earliest event you can use is the page Activate Event.

If you're using the grid within a secondary page, the runtime object for the grid isn't created until the secondary page is run. The grid object can't be obtained until then, which means that the earliest PeopleCode event you can use to activate a grid that's on a secondary page is the Activate event for the secondary page.

The attributes you set for displaying a page grid remain in effect only while the page is active. When you switch between pages in a component, you have to reapply those changes *every time* the page is displayed.

In addition, the Activate event associated with a page fires every time the page is displayed. Any PeopleCode associated with that Activate event runs, which may undo the changes you made when the page was last active. For example, if you hide a grid column in the Activate event, then display it as part of a user action, when the user tabs to another page in the component, then tabs back, the Activate event runs again, hiding the grid column again.

If a user at runtime hides a column of a grid, tabs to another page in the component, then tabs back to the first page, the page is refreshed and the grid column is displayed again.

When you place a grid on a page, the gird is automatically named the same as the name of the primary record of the scroll for the grid. This is the name you use with the GetGrid function. You can change this name on the Record tab of the Grid properties.

Note. There is no visible property for a grid, just a grid column. However, you can still hide an entire grid. Remember, many of the Rowset methods and properties work on a grid. If you want to hide an entire grid, get the rowset for that grid by using the HideAllRows Rowset class method.

See Also

PeopleCode Developer's Guide, PeopleCode and the Component Processor, Activate Event

Chapter 27, Rowset Class , HideAllRows, page 1493

Data Type for a Grid or Grid Column Object

Grids are declared using the Grid data type. For example,

```
Local Grid &MYGRID;
```

Grid Columns are declared using the GridColumn data type. For example:

```
Local GridColumn &MYGRIDCOL;
```

Scope of a Grid or Grid Column Object

Both the grid and grid column objects can be instantiated from PeopleCode only.

A grid is a control on a page. You generally use these objects only in PeopleCode programs that are associated with an online process, not in an Application Engine program, a message subscription, a Component Interface, and so on.

In addition, PeopleSoft builds a page grid one row at a time. Because the Grid class applies to a complete grid, you can't attach PeopleCode that uses the Grid class to events that occur before the grid is built; the earliest event you can use is the page Activate Event.

Grid Class Built-in Function

Chapter 1, PeopleCode Built-in Functions, GetGrid , page 246

Grid Class Method

In this section, we discuss the GetColumn Grid class method.

GetColumn

Syntax

GetColumn(*columnname*)

Description

The **GetColumn** method instantiates a grid column object from the GridColumn class, and populates it with a grid column from the grid object executing this method. Specify the grid column name in the page field properties for that field, consisting of any combination of uppercase letters, digits and "**#**", "**$**", "**@**", and "**_**".

To specify a grid column name:

1. Open the page in PeopleSoft Application Designer, select the grid and access the page field properties.
2. Select the Columns tab on the grid properties.
3. Either double-click on the grid column you want to name, or select the column and click the Properties button.
4. On the General tab, type the grid column name in the Page Field Name field.

Note. Although it's possible to base multiple grid columns on the same record field, the Page Field Name you enter for each grid column must be unique, not just for that grid, but for that page. If you have two grids on a page, every page field name must be unique to the page.

Parameters

Parameter	Description
columnname	Specify a string containing the value of the Page Field Name on the General tab of the grid column's properties. You must have previously entered a value for the Page Field Name for the grid column you want to work with.

Returns

A GridColumn object populated with the grid column specified as the parameter to this method.

Example

```
local Grid &MYGRID;
local GridColumn &MYGRIDCOLUMN;

&MYGRID = GetGrid(PAGE.EMPLOYEE_CHECKLIST, "EMPL_GRID");
&MYGRIDCOLUMN = &MYGRID.GetColumn("CHECKLIST_ITEMCODE");
```

The following function loops through rows on a grid. The function finds each row that is selected. It does this through the Selected property of the Row class of PeopleCode. Data is then moved from the selected row to a new row, on a different grid, in the same page. The way in which this function is written, data is moved from &SCROLL_SHELF to &SCROLL_CART. These are two different rowset objects, of two different grids, on the same page. Note that the two grids in this example are on the same occurs level.

```
/* Moving data between grids on the same occurs level */
/* of the same page */

Local Rowset &SCROLL_CART, &SCROLL_SHELF;

Function move_rows(&SCROLL_CART As Rowset, &SCROLL_SHELF As Rowset);

  &I = 1;
/* loop to find whether row is selected */
  Repeat
  If &SCROLL_SHELF.GetRow(&I).Selected = True Then

    If All(&SCROLL_CART(1).GetRecord(1).QEPC_ITEM.Value) Then

    &SCROLL_CART.InsertRow(&SCROLL_CART.ActiveRowCount);
    End-If;

  /* if it is selected move data to other grid   */

    &SCROLL_SHELF.GetRow(&I).GetRecord(1).CopyFieldsTo(&SCROLL_CART.GetRow(&SCROLL_⇒
CART.ActiveRowCount).GetRecord(1));

/* delete row from current grid so data disappears  */

    &SCROLL_SHELF.DeleteRow(&I);
    &I = &I - 1;
   End-If;

  &I = &I + 1;
  Until &I = &SCROLL_SHELF.ActiveRowCount + 1;

/* end of loop ************************************/

End-Function;

/********* end of function ***************************/

/* Creating the rowset object */

&SCROLL_CART = GetLevel0()(1).GetRowset(SCROLL.QEPC_CART);
&SCROLL_SHELF = GetLevel0()(1).GetRowset(SCROLL.QEPC_SHELF);

/* calling the function   */
```

```
move_rows(&SCROLL_CART, &SCROLL_SHELF);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetGrid , page 246.

Grid Class Properties

In this section, we discuss each Grid class property.

gridcolumn

Description

If a grid column name is used as a property, it accesses the grid column with that name. This means the following code:

```
&Mycolumn = &MyGrid.gridcolumnname;
```

acts the same as

```
&Mycolumn = &MyGrid.GetColumn(columnname);
```

This property is read-only.

Label

Description

This property returns a string specifying the label that appears as the title of the grid.

Note. You can't use this property to set labels longer than 100 characters. If you try to set a label of more than 100 characters, the label is truncated to 100 characters. Always put any changes to labels in the **Activate** event. This way the label is set every time the page is accessed.

This property is read-write.

GridColumn Class

Grid columns are page fields that comprise a grid, which is itself a page field. Similarly, a group of GridColumn objects comprises a Grid object. The GridColumn class enables you to change grid column attributes without having to write PeopleCode that loops through every row of the grid.

This class has no methods or built-in functions, only properties.

Note. To use a GridColumn object, you must instantiate it from a grid object. This requires you to first instantiate a Grid object using the GetGrid built-in function.

See Chapter 1, PeopleCode Built-in Functions, GetGrid , page 246.

GridColumn Class Properties

In this section, we discuss each GridColumn class property.

Enabled

Description

This property specifies whether the fields in the column are enabled (that is, can be edited) or if they are "disabled", that is, un-editable. All columns are enabled by default.

This property is read-write.

Example

```
If Not &ValidID Then
  &MYGRID = GetGrid(PAGE.EMPLOYEE_CHECKLIST, "EMPL_GRID");
  &MYGRIDCOLUMN = &MYGRID.GetColumn("CHECKLIST_ITEMCODE");
  &MYGRIDCOLUMN.Enabled = False;
End-If;
```

Label

Description

This property returns a string specifying the display label for the GridColumn object executing the property, as distinct from the grid column's Record Field Name and Page Field Name. This property overrides the equivalent settings on the Label tab of the grid column's Page Field Properties.

Note. You can't use this property to set labels longer than 100 characters. If you try to set a label of more than 100 characters, the label is truncated to 100 characters. If you change the column label, then change the field label, you may overwrite the column label with the field label. The grid object is part of a page, *not* the data buffers, while the field is part of the data buffers. To avoid this problem, always put any changes to column labels in the **Activate** event. This way the label is set every time the page is accessed.

This property is read-write.

Example

```
&MYGRIDCOLUMN.Label = "Checklist Item";
```

Name

Description

This property returns the name of the grid column, as a string. This value comes from the Page Field Name on the General tab in the Page Field Properties of the GridColumn object executing the property. This property was the value used to instantiate the GridColumn object.

This property is read-only.

Example

```
&PF_NAME = &MYGRIDCOLUMN.Name;
```

Visible

Description

This property returns a Boolean value which specifies whether a grid column should be hidden from view. Set this property to False to hide the grid column, and to True to unhide the grid column. This property defaults to True.

The **Visible** property also hides grid columns that are displayed as tabs in the PeopleSoft Internet Architecture.

If you specify "Show Column on Hide Rows" in PeopleSoft Application Designer, the column headers and labels of a grid display at runtime, even when the rest of the column is hidden. You can’t override this value using PeopleCode.

This property is read-write.

Note. In previous releases of PeopleTools, the methodology for hiding a grid column was to use the Hide function in a loop to hide each cell in the column, one row at a time. This method of hiding grid columns still works. However, your application may experience deteriorated performance if you continue to use this method.

Example

```
&MYGRIDCOLUMN.Visible = False;
```

The following example checks for the value of a field in every row of the grid. If that value is "N" for every row, the column is hidden.

```
&RS = GetRowset(Scroll.EX_SHEET_LINE);
&HIDE = True;
While (&HIDE)

  For &I = 1 To &RS.ActiveRowCount;
   &OUT_OF_POLICY = &RS(&I).EX_SHEET_LINE.OUT_OF_POLICY.Value;
   &NO_RECEIPT_FLG = &RS(&I).EX_SHEET_LINE.NO_RECEIPT_FLG.Value;
   If &OUT_OF_POLICY = "Y" Then
    &OUT_OF_POLICY_HIDE = False;
     &HIDE = False;
   End-If;
```

```
   If &NO_RECEIPT_FLG = "Y" Then
     &NO_RECEIPT_HIDE = False;
     &HIDE = False;
   End-If;
  End-For;

  If &HIDE = True Then
   &HIDE = False;
  End-If;

End-While;

If Not (&OUT_OF_POLICY_HIDE) Then
  GetGrid(Page.EX_SHEET_LINE_APV1, "EX_SHEET_LINE").OUT_OF_POLICY.Visible = True;

Else

 GetGrid(Page.EX_SHEET_LINE_APV1, "EX_SHEET_LINE").OUT_OF_POLICY.Visible = False;

End-If;

If Not (&NO_RECEIPT_HIDE) Then

  GetGrid(Page.EX_SHEET_LINE_APV1, "EX_SHEET_LINE").NO_RECEIPT_FLG.Visible = True;

Else

   GetGrid(Page.EX_SHEET_LINE_APV1, "EX_SHEET_LINE").NO_RECEIPT_FLG.Visible =⇒
 False;

End-If;
```

See Also

PeopleSoft Application Designer, Using Page Controls, Using.

CHAPTER 15

Internet Script Classes (iScript)

This chapter describes how an Internet Script (iScript) works in your application, how to create an iScript, and the Internet Script classes that you use in your PeopleCode. It also discusses the following topics:

- URL vs. URI.
- Web libraries.
- iScript security.
- When to use an iScript.
- Details of an iScript URL.
- The Generate functions.
- Error handling.
- Scope of the Internet Script classes.
- Data Types of the Internet Script classes.
- Internet Script classes built-in functions.
- Internet Script reference.

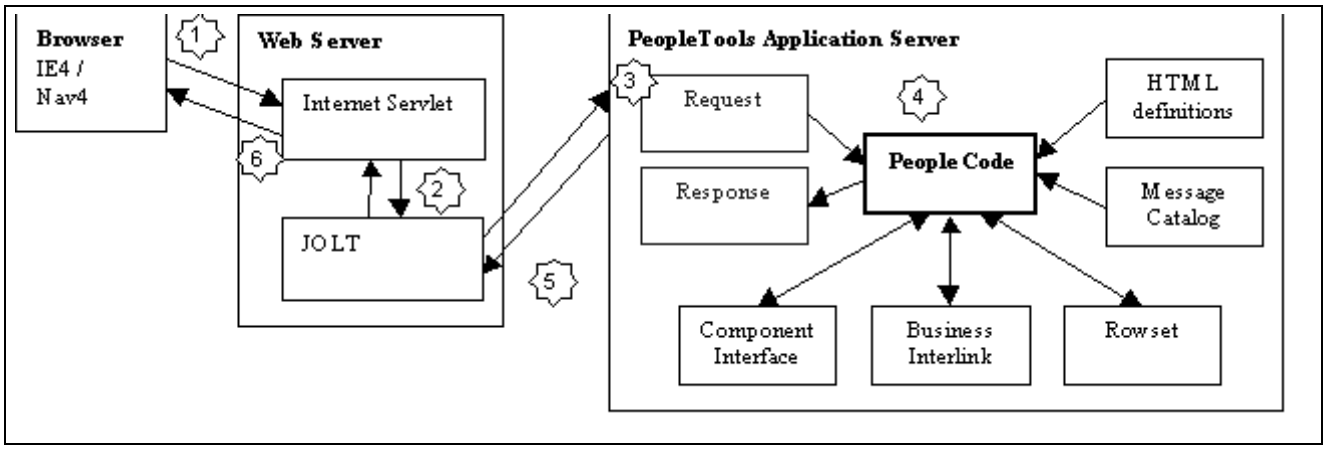
Understanding Internet Script Classes

An Internet Script, also called an iScript, is a specialized PeopleCode function that generates dynamic web content. iScripts interact with web clients (browsers) using a request-response paradigm based on the behavior of the Hypertext Transfer Protocol.

As a licensee of PeopleTools, you are licensed to use the base portal technology, which is limited to navigation to licensed PeopleSoft applications. If you want to register additional non-PeopleSoft content, customize your homepage, or create any pagelets, you must license PeopleSoft Enterprise Portal.

iScripts work with PeopleSoft Internet Architecture. You must have PeopleSoft Internet Architecture set up correctly before you can run an iScript.

The following flow chart shows the interaction of an iScript (PeopleCode) in a PeopleSoft Internet Architecture application.



Communication flow chart for Internet Client Scripts

1. An HTTP Request arrives at the web server specifying in its URL the Presentation Relay Servlet (psc). The path information included in the URL specifies which PeopleCode program to run. No component is specified because none is needed. The PeopleCode program (iScript) runs directly. The following is an example of the URL:

   ```
   http://serverx/Servlets/psp/ps84/e_procurement/fdm/s/WEBLIB_BEN_401k.PAGES.⇒
   FieldFormula.IScript_Home401k
   ```

2. The psc servlet is invoked by the web server. It serializes all necessary information from the Web Server Request and Response objects, then makes a call through JOLT to the PeopleSoft Application Server service passing the serialized object information.
3. The Application Server unpacks all of the object information and creates PeopleSoft versions of the Request and Response objects. It then calls the function in the PeopleCode program specified in the URL. The PeopleCode Request and Response objects are available to the PeopleCode program through the system variables %Request and %Response.
4. The PeopleCode program is then responsible for generating every aspect of the HTML page that is to be returned to the browser. It has access to the Request object to view items such as query string parameters, cookies, and headers. It also has access to Component Interfaces, Business Interlinks, and Rowset objects to interact with applications. HTML definitions are available to include language sensitive blocks of HTML. They can also contain JavaScript. The Message Catalog is also available for language sensitive messages. The Response object is used to "write" the generated HTML back to the browser.
5. When the PeopleCode program is finished running, the headers, cookies, and HTML in the PeopleSoft Response object are serialized and returned through JOLT to the Presentation Relay Servlet.
6. The Presentation Relay Servlet then unpacks the Response information generated by the PeopleCode, and plays it back to the web server response object, which in turn sends the response to the user's browser.

URL vs. URI

In this document, the term URL refers to the entire URL, that points to content. The following is an example of a URL:

```
http://someserver/servlets/psc/ps84/e_procurement/fdm/s/WEBLIB_Portal.⇒
```

```
FieldChange.IScript_DoSomething?page=view&key1=value1&key2=value2
```

A URI does not include the content information. Think of it as a subset of the URL that points to the location of the resource, but does not include any parameters passed to that resource. From the previous example, the URI portion of the URL is:

```
http://someserver/servlets/psc/ps84
```

Web Libraries

iScripts use the existing PeopleCode Function Library infrastructure. However, instead of naming your record FUNCLIB_xxx, all iScripts *must* be contained in records named WEBLIB_xxx. All of the existing tools and techniques for working with function libraries (such as Upgrade, Find In, Rename, and so on) also apply to Web Libraries.

See *PeopleCode Developer's Guide*, Using the PeopleCode Editor, Accessing PeopleCode External Functions.

To create an iScript:

1. Create or extend a domain specific WEBLIB (funclib).

   WEBLIBs are derived/work record definitions that have their name prefixed with WEBLIB_.

2. Add a function to a WEBLIB.

   The name of your function *must* be prefaced with **IScript_**. For example:

```
IScript_HelloWorld

IScript_FuncHRPage
```

Note. iScript functions take no arguments, and do not return a value.

The following iScript writes data.

```
Function IScript_HPDefaultCategories()

   &ClearDotImage = %Response.GetImageURL(Image.PT_PORTAL_CLEAR_DOT);
   &CatHTML = GetHTMLText(HTML.PORTAL_HP_CATEGORY, &ClearDotImage, ⇒
GetCategories());
   %Response.Write(&CatHTML);

End-Function;
```

The following iScript uses both the request and response objects to echo what the user types into an edit control.

Example iScript

```
Function IScript_HelloWorld()

%Response.WriteLine("<html><head><title>Hello World</title></head><body>");
   %Response.WriteLine("<h1>Hello World</h1>");
   %Response.WriteLine("<form method=POST action = " | %Request.FullURI | "?" |⇒
 %Request.QueryString | "><input type=submit value='Say this!'> <input type=⇒
edit name=WhatYouSaid value=" | %Request.GetParameter("WhatYouSaid") | "><⇒
/form>");

   rem echo back what the user typed into the edit box...;
   %Response.WriteLine("<p><font face='Arial, Helvetica' size='5' color='blue'⇒
 >You said: ");
   %Response.WriteLine(%Request.GetParameter("WhatYouSaid"));

   %Response.WriteLine("</body></html>");
   Return;
End-Function;
```

iScript Security

iScripts are secured on your system similar to Component Interfaces. After you create a WEBLIB record, and your functions, you must add them just as you would add a page to an application.

To add security to an iScript :

1. Navigate to the Maintain Security page and open the Permission List to which you want to give access.
2. Select Web Libraries.

   You may need to scroll through the tabs at the top of the page to access the Web Libraries tab.
3. (Optional) Add new WebLibrary for Permission List.

If this is a new Web Library, you must add it to the permission list. Click the plus button to add a new Web Library, then select the library you want to add from the drop-down list box.

4. Select **Edit** for the Web Library you want to grant access for.
5. Select the access you want to give each function in the WEBLIB record.

   From the page that displays, you can choose to allow or disallow access on a per function basis. The buttons on the side of the page either grant or disallow access for all functions.

See *PeopleTools PeopleBook: PeopleSoft Security*

When to Use an iScript

iScripts give you complete control over the HTML sent to the browser. This enables flexibility in creating a user interface. However, there are some responsibilities that you inherit when choosing to use iScripts.

- You are responsible for creating HTML and JavaScript that is cross-browser compatible.
- You are responsible for ensuring that all the text sent to the browser is stored in either the message catalog or in HTML definitions to enable translation of the text.
- iScripts aren't part of the regular Component Processor flow, so you are responsible for accessing the database in a multi-language, multi-market, and multi-currency sensitive way. (Do this by using a Component Interface to access the database.)

For these reasons, PeopleSoft recommends that you first try to build a page in PeopleSoft Application Designer. This approach enables PeopleTools to generate all of the HTML in a cross-browser, multi-language, multi-market, and multi-currency sensitive way.

You don't have to use an iScript to generate an entire page. You can use the Request and Response objects with an HTML area, so you can develop a portion of your HTML using the iScript objects, while allowing the majority of it to be generated by the Component Processor. If you use this method, instead of using the Response object's Write and WriteLine methods to output your HTML, you set the value of the HTML area to the HTML that you want to display. Remember, that just as with an iScript, you must ensure that the HTML you generate is cross-browser compatible and multi-market sensitive.

So when *should* you use iScripts? Here are two scenarios where iScripts would be an appropriate choice:

- The page being developed *cannot* be built using PeopleSoft Application Designer. An example of this is a page that requires more than one HTML form. PeopleSoft Internet Architecture places the entire page inside of a single form tag, so no other HTML form tags can be added. In this case the requirements of the page can't be met by pages created in PeopleSoft Application Designer, so use iScripts instead.

  Note. You should use Component Interfaces for all database access so you have multi-language and multi-currency sensitivity.

- The page being developed never accesses the database. Using a page and Component Processor for this type of page incurs unnecessary processing overhead. An example of this is a page that talks to another website and redisplays HTML from the remote site.

See *PeopleSoft PeopleBook Enterprise Portal*

Style Sheets and Styles

PeopleSoft recommends that developers using the iScripts always use styles (also known as classes) defined in the style sheets to specify the attributes (that is, background color, font, size, alignment, borders, and so on) of objects referenced in the iScripts. The Response object provides access to Style Sheets stored in the PeopleSoft database.

See Also

PeopleSoft Application Designer, Creating Style Sheet Definitions

Other Considerations

PeopleSoft does not recommend using the following technologies in iScripts:

- Java applets
- Browser plug-ins

Details of an iScript URL

Viewing an iScript requires the assembly of a URL with the following pieces:

Section	Description
http://Server/	The scheme (http / https) and web server name.
servlet_name/	The name of the physical servlet that the web server invokes to handle the request. This is either **psp** or **psc**.
SiteName/	The user-defined site name. This is defined during the installation of PIA. This enables you to set up multiple sites on one physical web server.
PortalName/	The name of the portal to use for this request. The portal object contains metadata that describes how to present the content (that is, template, pagelets and so on.)
NodeName/	The name of the node that contains the content for this request.
content_type/	The type of the content for this request. For iScripts, this is "s".

Section	Description
content_id	The identification of the content. This, and the type is the unique key to the content being retrieved.
?content_parm	The query string parameters (name value pairs) for the content.

For an iScript, the *content_id* has the following form:

```
Record.Field.Event.Function
```

The following is an example of the call to an iScript:

```
http://mlee2038//psp/PS84/e_procurement/fdm/s/WEBLIB_Portal.PORTAL_HEADER.⇒
FieldFormula.IScript_UniHeader_PIA
```

The Generate Functions

There are many different types of content on the portal, such as components, iScripts, homepages, and so on. Sometimes you want to access this content directly, through the URL that points to this content. To do this, PeopleSoft provides several Generate functions that generate the URL for the specified content.

The Generate functions can be categorized as:

- Content type.
- Absolute or relative URL.
- Portal service or content servlet (psp or psc).

The following is a list of the functions, plus a description of what to use them for. More information about a function is found under that function's description in the section PeopleCode Built-in Functions.

Function Name	Content Type	Absolute or Relative URL	Portal service or content servlet
GenerateActGuideContentUrl	Activity Guide	Absolute	psc
GenerateActGuidePortalUrl	Activity Guide	Absolute	psp
GenerateActGuideRelativeUrl	Activity Guide	Relative	NA
GenerateComponentContentURL	Component	Absolute	psc

Function Name	Content Type	Absolute or Relative URL	Portal service or content servlet
GenerateComponentContentRelURL	Component	Relative	psc
GenerateComponentPortalURL	Component	Absolute	psp
GenerateComponentPortalRelURL	Component	Relative	NA
GenerateComponentRelativeURL	Component	Relative	NA
GenerateExternalPortalURL	External	Absolute	psp
GenerateExternalRelativeURL	External	Relative	NA
GenerateHomepagePortalURL	Homepage	Absolute	psp
GenerateHomepageRelativeURL	Homepage	Relative	NA
GenerateQueryContentURL	WQuery	Absolute	psc
GenerateQueryPortalURL	Query	Absolute	psp
GenerateQueryRelativeURL	Query	Relative	NA
GenerateScriptContentURL	IScript	Absolute	psc
GenerateScriptContentRelURL	IScript	Relative	psc
GenerateScriptPortalURL	Iscript	Absolute	psp
GenerateScriptPortalRelURL	IScript	Relative	psp
GenerateScriptRelativeURL	IScript	Relative	NA
GenerateWorklistPortalURL	Worklist	Absolute	psp
GenerateWorklistRelativeURL	Worklist	Relative	NA

See Also

Chapter 1, PeopleCode Built-in Functions, page 1

Error Handling

All errors are handled through the session object. You should check the PSMessages collection to see if there are any errors in your code.

See Also

Chapter 29, Session Class , Error Handling , page 1560

Scope of the Internet Script Classes

The Request, Response, and Cookie classes can be accessed only from PeopleCode. They can be used only as part of a PeopleSoft Internet Architecture application—either as part of a page created in PeopleSoft Application Designer or an iScript. An iScript is always contained in a WEBLIB record. You wouldn't use any of these objects in an Application Engine program, a Component Interface, and so on. However, an iScript could call and start a Component Interface, a Business Interlink, and so on.

To access the Request or Response object, use the %Request and %Response system variable. These variables are objects, so you use them with dot notation.

```
%Response.SetContentType("text/HTML");

If %Request.GetParameter("encode") <> "" Then
```

Data Types of the iScript Classes

Cookie objects are instantiated from Response object. You can declare the cookie object as data type cookie before any functions. You can also declare variables of type Response and Request, to assign the %Response and %Request system variables to local variables.

```
Local Cookie &Cookie;
Local Response &Resp;
Local Request &Req;

Function IScript_SetCookie()
    rem set a cookie called "MyCookie" to store my user id in it's value.  Make⇒
 the cookie expire when the user's browser session expires;

   &Resp = %Response;
   &Req = %Request;
```

```
    &Cookie = &Resp.CreateCookie("MyCookie");
    &Cookie.Value = %UserId;
    &Cookie.MaxAge = - 1;

End-Function;
```

Internet Script Classes

The following is a description of the PeopleCode classes, methods, and properties that you use to create an iScript.

Request Class

The Request object encapsulates all information from the request issued from the browser. This includes the URI, Query String, QueryString parameters, Cookies, and Headers. The properties for the Request object are read-only.

Parameters

Request parameters are name-value pairs sent by the client to the web server as part of an HTTP request. They include the pairs in the query string part of the URL, and data posted in a form, if the request was a post request. Multiple parameter values can exist for any given parameter name. The following methods are available to access parameters:

- GetParameter
- GetParameterNames
- GetParameterValues

The GetParameterValues method returns an array of String objects containing all the parameter values associated with a parameter name. The value returned from the GetParameter method equals the first value in the array of String objects returned by GetParameterValues.

All form data from both the query string and the post body are aggregated into the request parameter set. The order of this aggregation is that query string data appears before post body parameter data.

Response

The response object encapsulates all the information to be sent back to the browser. This includes the body of the response, content type, response headers, and cookies.

The response object also includes helper methods that retrieve HTML, Image, StyleSheet, and other objects from the database and move them to the web server, and returning strings that can be used to reference these objects in the response content.

The body of the response is created using the response object’s Write and WriteLine methods. If the content type of the response is not explicitly set in the PeopleCode program, it defaults to "text/html".

Cookies

Cookies are data sent from the client to the server on every request that the client makes. Cookies are stored by the browser, either on disk or in memory, and returned to the server that originally set the cookie.

The Internet Script API makes available the name and value of each cookie that's sent with the request, through the Request object's GetCookieNames and GetCookieValue methods. The API also enables you to set new cookies (or alter existing cookies) through the Response object's CreateCookie method and the Cookie object's properties.

Internet Script Classes Built-in Functions

Chapter 1, PeopleCode Built-in Functions, GenerateActGuideContentUrl , page 198

Chapter 1, PeopleCode Built-in Functions, GenerateActGuidePortalUrl , page 199

Chapter 1, PeopleCode Built-in Functions, GenerateActGuideRelativeUrl , page 200

Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentRelURL , page 201

Chapter 1, PeopleCode Built-in Functions, GenerateComponentContentURL , page 203

Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalRelURL , page 205

Chapter 1, PeopleCode Built-in Functions, GenerateComponentPortalURL , page 207

Chapter 1, PeopleCode Built-in Functions, GenerateComponentRelativeURL , page 209

Chapter 1, PeopleCode Built-in Functions, GenerateExternalPortalURL , page 211

Chapter 1, PeopleCode Built-in Functions, GenerateExternalRelativeURL , page 212

Chapter 1, PeopleCode Built-in Functions, GenerateHomepagePortalURL , page 213

Chapter 1, PeopleCode Built-in Functions, GenerateHomepageRelativeURL , page 214

Chapter 1, PeopleCode Built-in Functions, GenerateQueryContentURL , page 218

Chapter 1, PeopleCode Built-in Functions, GenerateQueryPortalURL , page 220

Chapter 1, PeopleCode Built-in Functions, GenerateQueryRelativeURL , page 221

Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentRelURL , page 223

Chapter 1, PeopleCode Built-in Functions, GenerateScriptContentURL , page 224

Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalRelURL , page 226

Chapter 1, PeopleCode Built-in Functions, GenerateScriptPortalURL , page 227

Chapter 1, PeopleCode Built-in Functions, GenerateScriptRelativeURL , page 229

Chapter 1, PeopleCode Built-in Functions, GenerateWorklistPortalURL , page 231

Chapter 1, PeopleCode Built-in Functions, GenerateWorklistRelativeURL , page 232

Request Class Methods

In this section, we discuss the Request class methods. The methods are discussed in alphabetical order.

GetContentBody

Syntax

GetContentBody()

Description

This method retrieves the text content of an XML request. This is part of the incoming Business Interlink functionality, which enables PeopleCode to receive an XML request and return an XML response.

Parameters

None.

Returns

A string.

Example

The following example gets the XML text content of a request, then uses that as input to create a BiDoc. Then the BiDoc methods GetDoc and GetValue are used to access the value of the skills tag.

```
Local BIDocs &rootInDoc, &postreqDoc;
Local string &blob;
Local number &ret;

&blob = %Request.GetContentBody();
/* process the incoming xml(request)- Create a BiDoc */
&rootInDoc = GetBiDoc(&blob);
&postreqDoc = &rootInDoc.GetDoc("postreq");
&ret = &postreqDoc.GetValue("skills", &skills);
```

See *PeopleTools PeopleBook: PeopleSoft Business Interlink Application Developer's Guide*

GetCookieNames

Syntax

GetCookieNames()

Description

This method returns an array containing names of all the cookies present in this request. If there are no cookies in the request, an empty array is returned. When cookies are present, this method returns an array of strings.

GetCookieValue

Syntax

GetCookieValue (*name*)

Description

This method returns a string containing value of the cookie identified *name*. The *name* parameter takes a string value. The NPmatch between *name* and the request cookie is case-insensitive. If there is no cookie in the request matching the name, an empty string is returned.

GetHeader

Syntax

GetHeader(*name*)

Description

This method returns the value of the header requested by the string *name*. The match between *name* and the request header is case-insensitive. If the header requested does not exist, an empty string is returned.

Example

The following example gets the "Referer" header to see where the request came from:

```
&Referer = %Request.GetHeader("Referer");
```

GetHeaderNames

Syntax

GetHeaderNames()

Description

This method returns an array of Strings representing the header names for this request.

GetHelpURL

Syntax

GetHelpURL(*HelpContext*)

Description

This method returns the help context path (as a string) to the help directory. It's used to construct a URL to a specific help page.

Example

```
&HelpURLString = %Request.GetHelpURL("hc0110");
```

GetParameter

Syntax

GetParameter(*name*)

Description

This method returns the value of a specified query string parameter or posted form data parameter. The match between *name* and the request parameter is case-insensitive. If there are multiple parameter values for a single name, the value returned is the first value in the array returned by the **GetParameterValues** method. If the parameter has (or could have) multiple values, you should use the GetParameterValues method in your Internet scripts.

See Also

Chapter 15, Internet Script Classes (iScript) , GetParameterValues, page 944.

GetParameterNames

Syntax

GetParameterNames()

Description

This method returns all the parameter names for this request as an array of Strings, or an empty array if there are no input parameters.

GetParameterValues

Syntax

GetParameterValues(*name*)

Description

This method returns the values of the specified parameter (*name*) as an array of Strings, or an empty array if the named parameter does not exist. The *name* parameter takes a string value.

Request Class Properties

In this section, we discuss the Request class properties. The properties are discussed in alphabetical order.

AuthTokenDomain

Description

This property returns the web server domain as a string across which the single signon authentication token is valid. Use this property as the domain of any cookie which you want to apply across the same domain as the single signon token.

The value of this property is in the format:

```
".domain.com"
```

Note that it begins with a dot "." character. You can use this format as the value of a cookie's domain property.

However, sometimes you cannot use a value that is prefixed with a dot character. For example, the **javascript document.docmain** property should not begin with a dot. In this case, you must strip it off manually in your code.

Note. The value of this property is the domain across which the authentication token is valid, set in the **AuthTokenDomain** configuration property in the configuration properties file. The value of the system variable **%AuthenticationToken** is the authentication token itself.

This property is read-only.

See Also

Chapter 3, System Variables , %AuthenticationToken , page 564.

ByPassSignOn

Description

This property returns a Boolean value, indicating what **ByPassSignOn** has been set to in the configuration properties file. If this value has been set to True, this property returns True, False otherwise.

This property is read-only.

BrowserPlatform

Description

This property returns a string containing the value the web server passes as the browser platform.

This property is read-only.

BrowserType

Description

This property returns a string describing the browser that sent the request.

This property is read-only.

BrowserVersion

Description

This property returns a string describing the version of the browser that sent the request.

This property is read-only.

ContentURI

Description

This property returns a string containing the portion of the current URI before the portal name, and referencing the content servlet (psc).

For the URL:

```
http://localhost/psc/ps84/PORTAL/NODE/e_procurement/fdm/s/WEBLIB. . .
```

This property returns the following:

```
http://localhost/psc/ps84/
```

This property is read-only.

ExpireMeta

Description

This property returns the refresh meta-tag string that contains the **cmd=expire** parameter (the ExpireMeta string.)

The following is an example of the ExpireMeta string:

```
<meta HTTP-EQUIV='Refresh' Target='_top' CONTENT='10; URL=/servlets/psp/ps84/?cmd=⇒
expire'>
```

All Internet script pages should use this property to generate the meta tag to cause the page to expire. It is the same expiration tag used by pages originally created using PeopleSoft Application Designer, unless the Internet script generates a menu link in a separate frame that is not supposed to expire.

This tag should be included in the <head> section of the HTML generated by the iScript.

This property is read-only.

Example

```
%Response.Write("<html><head>");
%Response.Write(%Request.ExpireMeta);
%Response.Write("</head><body>");
```

FullURI

Description

This property returns the complete URI up to, but not including, the query string.
This method returns a string value.

To return only the request URI (that is, without the scheme, server name or port) use the **RequestURI** method.

This property is read-only.

Example

From the following code

```
&FullURI = &Request.FullURI;
```

&FullURI might contain the following:

```
http://serverx/servlets/psp/ps84/Portal/Node/s/WEBLIB_TEST.SCRIPTS.⇒
FieldFormula.IScript_Test
```

See Also

Chapter 15, Internet Script Classes (iScript) , RequestURI, page 948.

HTTPMethod

Description

This property returns the HTTP method as a string, (for example, GET, POST, PUT) by which this request was made.

This property is read-only.

LogoutURL

Description

This property returns the complete URL (as a string) to logout of the PeopleSoft session. Use this property to generate a link that causes a page to logout. You should normally not need to include this link on your page.

This property is read-only.

PathInfo

Description

This property returns any path information following the servlet path, but prior to the query string. This property returns null if there is no path information following the servlet path.

This property is read-only.

Example

For the URL:

```
http:/localhost/psp/ps84/PORTAL/NODE/e_procurement/fdm/s/WEBLIB. . .
```

This property returns the following:

```
psp84/PORTAL/NODE/eprocurements/fdm/s/WEBLIB_. . .
```

Protocol

Description

This property returns the protocol being used for this request as a string in the following form:

```
protocol/major_version.minor_version
```

An HTTP 1.0 request, as defined by the HTTP 1.0 specification, should return the string HTTP/1.0. Use the Scheme property instead of the Protocol property when generating hrefs (links).

This property is read-only.

QueryString

Description

This property returns the query string present in the request URL, if any. A query string is defined as any information following a ? character in the URL.

If there is no query string, this method returns null.

Use the request Parameters methods (**GetParameterNames**, **GetParameterValues**, **GetParameter**) to get the values of individual parameters on the query string.

This property is read-only.

RelativeURL

Description

This property returns whether a relative URL should be generated for PeopleSoft Internet Architecture pages. This property is set in the configuration properties file. This property takes a Boolean value: True if a relative URL is generated for PeopleSoft Internet Architecture pages, False otherwise.

This property is read-only.

RemoteAddr

Description

This property returns the IP address of the agent that sent the request.

This property is read-only.

RemoteHost

Description

This property returns the fully qualified host name of the agent that sent the request.

This property is read-only.

RequestURI

Description

This property returns the URI, without the protocol.

This property is read-only.

Example

This example uses the following URL:

```
http://serverx/servlets/psc/ps84/PORTAL/NODE/fdm/s/WEBLIB_Portal. . .
```

From the following code

```
&MyRequestURI = &Request.RequestURI;
```

&MyRequestURI contains the following:

```
/servlets/psc/ps84/PORTAL/NODE/fdm/s/WEBLIB_PORTAL . . .
```

Use the following to build an absolute URL:

```
&myURL = &Request.Scheme | "://" | &Request.ServerName | ":" | &Request.Port | ⇒
&Request.RequestURI | "?" | &Request.QueryString;
```

Scheme

Description

This property returns the scheme, also known as the protocol, used by the request. Common schemes include **http**, **https**, **ftp**, and **telnet**. This property returns a string value.

This property is read-only.

Example

For the URL

```
http://serverx/servlets/psp/ps84/eprocurement/fdm/s/WEBLIB. . . .
```

the Scheme property returns:

```
http
```

ServerName

Description

This property returns the host name of the server on which the servlet is running. This property returns a string value.

This property is read-only.

Example

For the URL

```
http://serverx/servlets/psp/ps84/eprocurement/fdm/s/WEBLIB. . . .
```

the ServerName property returns:

```
serverx
```

ServerPort

Description

This property returns an integer representing the port on which the browser's request was received (that is, the port on which the server is listening.)

This property is read-only.

Example

For the URL

```
http://servername:80/servlets/iclientservlet/peoplesoft8/?ICType=Panel. . .
```

the ServerPort property returns:

```
80
```

Timeout

Description

This property returns an integer representing the timeout value, in seconds, set in the configuration.properties file.

This property is read-only.

Response Class

The response object encapsulates all information to be returned from the iScript to the browser.

Response Class Methods

In this section, we discuss the Response class methods. The methods are discussed in alphabetical order.

Clear

Syntax

```
Clear()
```

Description

This method removes all cookies and headers as well as deletes any HTML that has been written to the Response object. After using this method, the Response object appears as it did when the script was first called.

CreateCookie

Syntax

CreateCookie(*name*)

Description

This method adds a cookie to the response with the name specified by the string *name*. It returns a reference to the cookie object that is used to update the values of this cookie. If the cookie by the specified name already exists, a reference to the existing cookie is returned. This method can be called multiple times to set more than one cookie. Cookie names may not contain the "$" character.

Note. The web server only allows 20 cookies per server, and randomly throws away some cookies when you exceed that number. PeopleTools uses 7 cookies, and some PeopleSoft applications use the remaining 13. You may or may not be able to create your own cookies to modify an existing PeopleSoft application.

GetCookie

Syntax

GetCookie(*name*)

Description

This method returns the cookie object specified by the string *name*. If the cookie is not already present in this response, a null value is returned.

GetCookieNames

Syntax

GetCookieNames()

Description

This method returns an array of strings that contains the names of all the cookies present in this response. If there are no cookies in the response, an empty array is returned.

GetHeader

Syntax

GetHeader(*name*)

Description

This method returns the value of the response header requested by the string *name*. The match between *name* and the response header is case-insensitive. If the header requested does not exist, an empty string is returned.

GetHeaderNames

Syntax

GetHeaderNames()

Description

This method returns an array of Strings representing the header names for this response.

GetImageURL

Syntax

GetImageURL(**IMAGE.***ImageName* | *rowset.row.record*)

Description

This method returns a string which represents the URL of the requested image, for images in the database.

Because the image field is a long data type, you can have only one on the record. Therefore you must specify the record only. The field on the record is assumed to be the image.

Parameters

Parameter	Description
imagename	Specify the name of an existing image. This name must be prefaced with the reserved word **IMAGE.**
rowset.row.record	Specify the path to the image, that is, the rowset, row, and record.

Example

The following example gets the URL to use for the image, and the second uses the URL reference.

```
&strImage = %Response.GetImageURL(Image.ARROW);

%Response.WriteLine("<img src=" | &strImage | ">");
```

The following gets an image from the record.

```
%Response.GetImageURL(&rs.GetRow(1).GetRecord(Record.EMPL_PHOTO));
```

GetJavaScriptURL

Syntax

GetJavaScriptURL(**HTML**.*HTMLname*)

Description

This method returns a string which represents the URL of the requested HTML definition, for JavaScript HTML definitions stored in the database.

Example

This example assumes the existence in the database of an HTML definition called "HelloWorld_JS", that contains some JavaScript.

```
Function IScript_TestJavaScript()
```

```
   %Response.WriteLine("<script src= " | %Response.GetJavaScriptURL(HTML.Hello⇒
World_JS) | "></script>");

End-Function;
```

In this example, the HTML definition is called FOO. The following PeopleCode example:

```
%Response.GetJavaScriptURL(HTML.FOO)
```

Resolves to something similar to:

```
http://myserver/cache/js/FOO_ENG_1.js
```

See Also

PeopleSoft Application Designer, Creating HTML Definitions.

GetStyleSheetURL

Syntax

GetStyleSheetURL(STYLESHEET.*stylename*)

Description

This method returns a URL string pointing to a style sheet created in PeopleSoft Application Designer.

Example

In the following example, the first line gets the URL to use for the style sheet, and the second is the standard HTML to include the style sheet in the HTML document. The stylesheet link must be included in the HTML *before* any of the styles in the stylesheet are used. It should be included in the <head> section of the HTML document.

```
&strStyleSheet = %Response.GetStyleSheetURL(StyleSheet.BENEFITS_STYLE);

%Response.WriteLine("<link rel=stylesheet href=" | &strStyleSheet | " type=""text⇒
/css"">");
```

Now to use a style, assign the class attribute of your choice to your HTML tags, as follows:

```
%Response.WriteLine("<p class=PSTEXT>I am some classy text!!</p>");
```

RedirectURL

Syntax

RedirectURL(*location*)

Description

This method sends a temporary redirect response to the client using the location specified by the string location. The given location must be an absolute URL. Relative URLs are not permitted. No further output should be made by the iScript after calling this method.

Example

```
%Response.RedirectURL(&URL);
```

SetContentType

Syntax

SetContentType(*Type*)

Description

This method sets the content type for this response. The parameter *type* takes a string value. This type may later be implicitly modified by the addition of properties such as the MIME charset property if the service finds it necessary and the appropriate property has not been set.

The content type defaults to "text/html" if it is not set.

SetHeader

Syntax

SetHeader(*name, value*)

Description

This method sets a response header with the specified name and value. Both parameters take string values. If the field has already been set with a value, this new value overwrites the previous one.

Example

The PortalRegisteredURL response header identifies to the portal the registered URL of the current content, so that it can find the correct content reference to look at for the template.

You can override the value of the PortalRegisteredURL response header in a PIA Script or PIA Page by adding the header to the response yourself, like this:

```
%Response.SetHeader("PortalRegisteredURL", &myURL);
```

You can do this to register the content with additional parameters.

UseSimpleURL

Syntax

UseSimpleURL({**True** | **False**})

Description

The **UseSimpleURL** method sets the response header that informs the portal that the current content uses relative simple URLs, and does not have to be proxied.

By default, all PeopleSoft Internet Architecture content sets the header to true. You must to call this method to specify that your content does not use the simple URL format, and therefore does need to be proxied.

Parameters

Parameter	Description
True | **False**	Specify a Boolean value, either True or False, to indicate whether simple URLs are used.

Returns

None.

Write

Syntax

Write(*String*)

Description

This method prints *String* to the HTTP output stream.

You can use an HTML string from PeopleSoft Application Designer HTML catalog with the **Write** method if you also use the **GetHTMLText** function, as follows:

```
%Response.Write(GetHTMLTEXT(HTML.MY_HTML));
```

You can also use an XML string. The following example takes a BiDocs structure that contains an XML response and puts that into a text string. After this is done, the %Response.Write function can send this as an XML response.

```
Local BIDocs &rootDoc;
Local string &xmlString;

&xmlString = %Response.GetContentBody();
&rootDoc = GetBIDoc(&xmlString);

/* do processing */

&xmlString = &rootDoc.GenXMLString();
%Response.Write(&xmlString);
```

WriteLine

Syntax

WriteLine(*String*)

Description

This method adds a carriage control and line feed to the end of the string *String*, then prints the string to the HTTP output stream.

You can use an HTML string from PeopleSoft Application Designer HTML catalog with the **WriteLine** method if you also use the **GetHTMLText** function, as follows:

```
%Response.WriteLine(GetHTMLTEXT(HTML.MY_HTML));
```

Response Class Property

In this section, we discuss the Response class property.

Charset

Description

This property returns the character set used by the response object as a string. This value is set in the user personalizations.

This property is read-only.

See Also

PeopleTools Security, Managing PeopleSoft Personalizations.

Cookie Class

The cookie class encapsulates all information to be sent to the browser for a single cookie. The cookie class is used by the response object to set new cookies. The request object uses a simple name-value pair mechanism for retrieving previously stored cookie values sent with the HTTP request.

Cookie Class Properties

In this section, we discuss the Cookie class properties. The properties are discussed in alphabetical order.

Domain

Description

This property returns the domain of this cookie, or null if not defined. This property is a string value.

This property is read-write.

Example

```
&cookie = &Response.AddCookie("My cookie", "My value");

&cookie.Domain = ".MyDomain.com";
```

MaxAge

Description

This property represents the maximum specified age of the cookie, as a signed number in seconds. The default value is -1. Setting the MaxAge property to a negative value ensures the cookie does not persist on the client when the client session ends. If the MaxAge property is set to zero, the cookie is deleted immediately from the client.

Value	Description
Non Negative Integer	Lifetime of the Cookie in seconds
0	Delete the cookie from the client immediately
-1	Default MaxAge property value (remove the cookie after the client exits)
Negative Integer	Remove the cookie after the client exits

This property is read-write.

Name

Description

This property returns the name of the cookie as a string.

This property is read-only.

Path

Description

This property returns the prefix of all the URL paths for which this cookie is valid, or an empty string if not defined.

This property is read-write.

Secure

Description

This property specifies whether the cookie is to be secured. This property takes a Boolean value. The default value is False. Secure cookies are sent only if the client has established a secure link (such as HTTPS) with the server. This property is read-write.

Value

Description

This property returns the value of the cookie (as a string), or an empty string if it isn't defined.

This property is read-write.

CHAPTER 16

Java Class

This chapter provides an overview of Java class and discusses the following topics:

- Supported versions of Java.
- Naming standards for classes and packages.
- System setup for Java classes.
- From PeopleCode to Java.
- From Java to PeopleCode.
- PeopleCode and Java data types mapping.
- Considerations when using the PeopleCode Java functions.
- Error handling and the PeopleCode Java functions.
- The Java debugging environment.
- Java object declaration.
- Scope of a Java object.
- PeopleCode Java built-in functions.

Understanding Java Class

Using the PeopleCode Java functions, you can access Java classes, or create instances of Java objects. Calling a Java program from PeopleCode can greatly extend your application. Also, the PeopleCode integration with Java enables you to pass parameters such as record, record fields, or rowsets into Java programs.

The PeopleCode Java functions are:

- CreateJavaArray
- CreateJavaObject
- GetJavaClass

See Also

http://java.sun.com/index.html

Supported Versions of Java

PeopleSoft supports Java version 1.3.1 for all its platforms. The Java Runtime Engine (JRE) comes bundled with PeopleSoft for the following platforms:

- Sun
- AIX
- HP
- NT

On other platforms, you may have to install the JRE.

See your platform installation documentation for more information.

Naming Standards for Classes and Packages

PeopleSoft recommends following Sun standards for naming classes and packages.

Java classes delivered with PeopleSoft are generally in the following directory:

```
<PS_HOME>\appserv\classes
```

At PeopleSoft, we use the following for the package hierarchy:

```
com.peoplesoft.<prodline>.<prodcode>
```

Where *prodline* is the company standardized product line code (like hrms) and *prodcode* is the product code unique within the product line (like hr).

In addition, if there are any classes that are common across products, they are placed into the common package for that product line, that is:

```
com.peoplesoft.<prodline>.common
```

See Also

http://java.sun.com/docs/codeconv/index.html

System Setup for Java Classes

If you only access the classes that come defined with PeopleSoft you don't need to do any additional setup.

If you include your JAR or class files in the Java directory you don't need to do any additional setup. That is, you don't need to do anything if your files are in the following directory:

```
<PS_HOME>\appserv\classes
```

If you create your own Java classes that you want to access with PeopleCode, and you do not store them in the Java directory, you *must* create a PS_CLASSPATH environment variable that points to where your classes are stored. The exact path depends on how and where you package your classes.

- If you package your Java classes in a JAR file, and you do *not* place your JAR file in the Java class directory, you must specify the JAR file name (with the JAR extension) in the PS_CLASSPATH environment variable.

  Usually when there is a package hierarchy, the JAR file is at the package root, and contained within it a directory-like structure of the class files. For example, if the package hierarchy is like this:

  ```
  c:\myjava\com\mycompany\myprod\xxx.java
  ```

  Then you would have to set your PS_CLASSPATH environment variable to the following:

  ```
  c:\myjava\com.mycompany.myprod.jar
  ```

  Note. PeopleSoft recommends using the utility that comes with the Java SDK for creating JAR files.

- If you don't store the Java classes you create in a JAR file, and you do *not* place your class files in the Java directory, the PS_CLASSPATH environment variable must point to the top level of your package hierarchy.

  For example, suppose your Java classes were in the following package:

  ```
  c:\myjava\com\mycompany\myproduct\
  ```

  You would set the PS_CLASSPATH environment variable to:

  ```
  c:\myjava
  ```

Like most environment variables, you can specify more than one entry. On Windows, the PS_CLASSPATH entries are separated by semicolons. On Unix, they're separated by colons. The following is for Windows:

```
c:\myjava;d:\myjava\com\mycompany\myproduct\myjar.jar; . . .
```

The following would be for Unix:

```
/etc/myjava:/home/me/myjava/com/mycompany/myproduct/myjar.jar: . . .
```

This PS_CLASSPATH setting must be set for the process that runs the Java. For example, if Java is being called from PeopleCode that is running on the Application Server, it is the Application Server process that needs this setting. Similarly, if the Java is being run from PeopleCode in the Application Engine, that run of the Application Engine must have this setting.

Also when developing your own classes you must be aware that most Java virtual machines cache the class definitions. This means that even if you change the class files, a running Java virtual machine (inside, perhaps, an Application Server) that has loaded and is running the old version of the class files. It won't pick up the new version of the class files. You must restart the Application Server to make it reload the new classes.

See your system documentation for more information about setting up an environment variable.

From PeopleCode to Java

In this section, we provide an overview of state management concerns and present the following examples:

- State Management Concerns.
- CreateJavaObject example.
- CreateJavaArray example.
- GetJavaClass example.

State Management Concerns

The application server is *stateless*, which means that it doesn't keep any information (state) for its clients between calls to it. For one reason, calls to the application server can use different actual servers for different calls. When you are using Java in the application server, be careful to not leave state in the Java virtual machine that would cause your application to fail if a different application server (which would use a different invocation of the Java virtual machine) was used for subsequent calls. One method to leave state in the virtual machine is to use static (class) variables.

Similar considerations to these apply using Java in Application Engine programs, though here the difficulty arises when you try to checkpoint and then restart the program. The restart starts with a Java virtual machine invocation that doesn't have any of the state you might have stored into the Java virtual machine before the checkpoint.

JavaObject variables cannot have Global or Component scope because of this lack of ability to save the state of these objects.

An example of this is issuing messages. When you're running with PeopleSoft Internet Architecture and issue a message, the message is produced by an end-user action, so the Application Server gathers up its state to return it to the browser. This state saving attempts to save the current PeopleCode execution state, causing it to issue an error because of the JavaObject.

The solution is to not have any non-Null JavaObject objects when the message is issued.

The following is a simple Java program:

```
public class PC_Java_Test{

 public String pcTest(){

  String message;

  message = "PeopleCode is successfully executing Java.";

  return message;
 }
}
```

Here is the PeopleCode that calls this Java program. Note that the JavaObject is set to NULL before the message is issued.

```
&java_test = CreateJavaObject("PC_Java_Test");
&java_message = &java_test.pcTest();
&java_test = Null;
WinMessage(&java_message);
```

In SavePreChange, Workflow, or SavePostChange PeopleCode the situation is more complicated. Usually messages with a zero style parameter (no buttons other than OK and perhaps Explain, therefore no result possible except OK) are queued up by the Application Server. They are output by the browser when the service completes, so the serialization won't happen until after the PeopleCode has finished, so you won't have to set your JavaObject to NULL. With other kinds of messages, you must do this.

CreateJavaObject Example

The following is an example program creating a Java object from a sample program that generates a random password.

```
/* Example to return Random Passwords from a Java class */
Local JavaObject &oGpw;

/* Create an instance of the object */
&oGpw = CreateJavaObject("com.PeopleSoft.Random.Gpw_Demo");

&Q = "1";

/* Call the method within the class */
&NEW_VALUE = &oGpw.getNewPassword(&Q, PSRNDMPSWD.LENGTH);

/* This is just returning one value for now */
PSRNDMPSWD.PSWD = &NEW_VALUE;
```

CreateJavaArray Example

Suppose we had a PeopleCode array of strings (&Parms) that we wanted to pass to a Java method xyz of class Abc. This example assumes that you don't know when you write the code just how many parameters you will have.

```
Local JavaObject &Abc, &RefArray;
Local array of String &Parms;

&Parms = CreateArray();

/* Populate array how ever you want to populate it */

&Abc = GetJavaObject("com.peoplesoft.def.Abc");

/* Create the java array object. */

&JavaParms = CreateJavaArray("java.lang.String[]", &Parms.Len);

/* Populate the java array from the PeopleCode array. */

&RefArray = GetJavaClass("java.lang.reflect.Array");
```

```
For &I = 1 to &Parms.Len
 &RefArray.set(&JavaParms, &I - 1, &Parms[&I]);
End-For;

/* Call the method. */
&Abc.xyz(&JavaParms);
```

GetJavaClass Example

The following example gets a system class.

```
&Sys = GetJavaClass("java.lang.System");
&Sys.setProperty("java.security.policy", "C:\java\policy");

WinMessage("The security property is: " | &Sys.getProperty("java.security.policy"));

&Props = &Sys.getProperties();
&Props.put("java.security.policy", "C:\java\policy");
&Sys.setProperties(&Props);

WinMessage("The security property is: " | &Sys.getProperty("java.security.policy"));
```

From Java to PeopleCode

The Java classes delivered with PeopleTools enable you to call PeopleCode from your Java program. Calling into PeopleCode works only from Java code that you have initially called from PeopleCode.

You must call PeopleCode facilities only from the same thread that was used for the call into Java. PeopleTools is not multithreaded.

You cannot call any PeopleCode facility that would cause the server to return to the browser for an end-user action, because the state of the Java computation cannot be saved and restored when the action is complete.

See Also

Chapter 16, Java Class, Considerations When Using the PeopleCode Java Functions, page 969

SysVar Java Class

Use the SysVar Java Class to refer to System Variables, such as %Language or %DBType.

For example, %Session, becomes SysVar.Session()

See Also

Chapter 3, System Variables , page 563

SysCon Java Class

Use the SysCon Java Class to refer to system constants, such as %SQLStatus_OK or %FilePath_Absolute.

For example, %CharType_Matched becomes SysCon.CharType_Matched.

Func Java Class

Use the Func Java Class to refer to built-in functions, such as CreateRowset or GetFile.

For example, SetLanguage(LANG_CD) becomes Func.SetLanguage(LANG_CD)

Name Java Class

The Name Java Class enables you to use the PeopleSoft reserved item references. This enables you to reference pages, components, records, fieldnames, and so on.

For example, in PeopleCode you can refer to a record field using the following:

```
recname.fieldname
```

With the Name class, you can use a similar construct:

```
new PeopleSoft.PeopleCode.Name("RECNAME", "FIELDNAME");
```

Note that these must be in the exact case as the item. As all PeopleTools items are named in uppercase, that means you must use uppercase.

As another example, in PeopleCode you can refer to a page using the following:

```
PAGE.pagename
```

In Java, it would be:

```
new PeopleSoft.PeopleCode.Name("PAGE", "PAGENAME");
```

Using PeopleCode Objects

The existing PeopleCode classes (like Array, Rowset, and so on) have properties and methods you can access.

- PeopleCode classes have the same names, so Record becomes Record, SQL becomes SQL, and so on.
- Methods are accessed by the method name.
- The name of a property is pre-pended with either **get** or **set**, depending on whether you're reading or writing to the property.

  For example, to get the IsChanged property would be getIsChanged. To set the value for a field would be &MyField.setValue.

Here is an example of a Java program that uses PeopleCode objects to access the database:

```
/*
 * Class Test
 *
 * This code is used to test the Java/PeopleCode interface.
 *
```

```
 */

import PeopleSoft.PeopleCode.*;

public class Test {

/*
 * Test
 *
 * Add up and return the length of all the
 * item labels on the UTILITIES menu,
 * found two different ways.
 *
 */

public static int Test() {
/* Get a Rowset to hold all the menu item records. */
Rowset rs = Func.CreateRowset(new Name("RECORD", "PSMENUITEM"), new Object[]{});
String menuName = "UTILITIES";
int nRecs = rs.Fill(new Object[]{"WHERE FILL.MENUNAME = :1", menuName});

int i;
int nFillChars = 0;
for (i = 1; i <= rs.getActiveRowCount(); i++) {
String itemLabel = (String)rs.GetRow(i)
.GetRecord(new Name("RECORD", "PSMENUITEM"))
.GetField(new Name("FIELD", "ITEMLABEL"))
.getValue();
nFillChars += itemLabel.length();
}

/* Do this a different way - use the SQL object to read each menu
   item record. */

int nSQLChars = 0;
Record menuRec = Func.CreateRecord(new Name("RECORD", "PSMENUITEM"));
SQL menuSQL = Func.CreateSQL("%SelectAll(:1) WHERE MENUNAME = :2",
new Object[]{menuRec, menuName});

while (menuSQL.Fetch(new Object[]{menuRec})) {
String itemLabel = (String)menuRec
.GetField(new Name("FIELD", "ITEMLABEL"))
.getValue();
nSQLChars += itemLabel.length();
}

return nFillChars + 100000 * nSQLChars;
}
}
```

```
This can be run from PeopleCode like this:
Local JavaObject &Test;
Local number &chars;

&Test = GetJavaClass("Test");
&chars = &Test.Test();

&Test = Null;
WinMessage("The character counts found are: " | &chars, 0);
```

Using Application Classes From Java to PeopleCode

You call a Java program from an Application Class the same way you do using any other PeopleCode program, that is, by using one of the existing Java class built-in functions.

Calling an Application Class from a Java program has the following considerations:

- Application Classes must be accessed using the object built-in functions, such as CreateObject, ObjectDoMethod, ObjectGetProperty, and so on.
- You cannot declare a variable of type HR.Package.SomeClass in your Java program. The variable must be of type Object.
- There is no pre-pending the word ‘get’ or ‘set’ for properties. All classes, methods, and properties are passed as strings.

The following is an example of how to call an Application Class from a Java program.

This is the Java program:

```
package com.peoplesoft.pcode;
import PeopleSoft.PeopleCode.*;

public class foo {

  public foo() {
  }
  public String getString() {

    Object foo = Func.CreateObject("GTP:Foo", new Object[]{});
    return (String)Func.ObjectDoMethod((Peer)foo, "GetString", new Object[]{});
  }
}
```

The following is the Application Class Foo, in the Application Package Foo:

```
class Foo
   method GetString() Returns string;
end-class;

method GetString
```

```
   /+ Returns String +/
   Return "Hello";
end-method;
```

The following is the PeopleCode program that starts it all:

```
Local JavaObject &foo = CreateJavaObject("com.peoplesoft.pcode.foo");
GTP_PARSER.GTP_STR_RESULT = &foo.getString();
```

PeopleCode and Java Data Types Mapping

The following table describes the matching of types for resolution of overloaded Java methods and basic conversions. The first Java Type/Class is the one that is produced in the absence of any other type of information.

PeopleCode Type	Java Type/Class
Float	double, float
Number	double, float, byte, char, short, int, long
Integer	int, byte, char, short, long
Boolean	Boolean
String	java.lang.String
Date	java.sql.Date
Time	java.sql.Time
Date Time	java.util.Date
any kind of object	any kind of object

The following table represents the conversions done to produce the Java class java.lang.Object. In addition to these, the conversions (listed in the previous table) from String onwards are done to produce a java.lang.Object.

PeopleCode Type	Java Type/Class
Float, Number	java.lang.Double
Integer	java.lang.Integer
Boolean	java.lang.Boolean

The following table represents other conversions that are done as required by the signature of a Java method or constructor.

PeopleCode Type	Java Class
Integer, Number, Float	java.lang.Integer, java.lang.Byte, java.lang.Character, java.lang.Short, java.lang.Long, java.lang.Float, PeopleSoft.PeopleCode.intHolder, PeopleSoft.PeopleCode.doubleHolder
String	PeopleSoft.PeopleCode.StringHolder
peoplecode builtin class Xxx	PeopleSoft.PeopleCode.Xxx
JavaObject	corresponding Java object

Considerations When Using the PeopleCode Java Functions

Some PeopleCode built-in functions can't be called from a Java program. Many of these restrictions arise because you can't serialize Java objects. Inside Java, you can't serialize and save the state of the Java Virtual Machine (JVM.)

This means that you cannot call the following built-in functions:

- Think-time functions, such as DoCancel, Prompt, RevalidatePassword, and so on.
- Math functions, such as Abs, Cos, and Sin. Use the Java math functions instead.
- Java class functions, such as CreateJavaArray, CreateJavaObject, or GetJavaClass.
- Deprecated functions, such as SetNextPanel, PanelGroupChanged, and so on. Use the new function instead, like SetNextPage, ComponentChanged, and so on.
- PeopleCode language elements, such as Exit, Return, If, and so on.
- Cursor position, such as SetCursorPos.
- Functions that rely on specific character encoding, such as char, code, exact, codeb, and so on.

- Menu appearance functions, such as CheckMenuItem, HideMenuItem, and so on.
- Transfer functions such as Transfer or TransferPage.
- DoSaveNow isn't allowed. However, DoSave is.

When you're creating your Java program, keep the following points in mind:

- If you're starting from an online application, avoid calling anything that will wait a long time.
- If you use third-party Java that require third-party platforms, you are limiting your program to just those platforms.

See Also

PeopleCode Developer's Guide, Using Methods and Built-In Functions, Think-Time Functions

Error Handling and the PeopleCode Java Functions

Java functions throw exceptions to indicate something unusual has happened. However, all exceptions from Java called by PeopleCode are turned into fatal errors. The PeopleCode program is terminated, and the user transaction must be canceled.

PeopleSoft recommends that you write a Java wrapper to handle errors.

Use either the All or None built-in functions to check values that are returned if you think you may call a Java method that is defined to return a string, but returns a Null object reference instead. Java Null object references are automatically converted into PeopleCode Null object references.

The Java Debugging Environment

To use Sun's JPDA V1.0 debugging architecture, you must do the following. These instructions are general: how you actually set up the debugger depends on your system.

To use the Java Debugging Environment (JDB):

1. Download and install a copy of JPDA V1.0.
2. Set the path for your system.

   Suppose you install JPDA V1.0 in C:\jpda. Set your PATH environment variable to include C:\jpda\bin.
3. Set the path for the application server.

   For the application server, set the Domain Settings/Add to PATH to include C:\jpda\bin.
4. Set the JavaVM Options.

   Set the JavaVM Options to be something like the following (see the JPDA documentation for a more complete example):

```
-Xdebug -Djava.compiler=NONE -Xnoagent -Xrunjdwp:transport=dt_socket,suspend=⇒
n,address=8765,server=y
```

5. Run the debugger.

   After starting the tools session and causing it to start the JVM, you can use the JDB command line debugger that comes with JPDA, using a command like the following:

   ```
   jdb -connect com.sun.jdi.SocketAttach:port=8765
   ```

   You can also use the (no cost) Forte for Java Community Edition IDE from Sun or any of the Java IDEs noted on the JPDA pages.

See Also

http://java.sun.com/products/jpda

http://www.sun.com/forte/ffj/ce/index.html

Data Type of a Java Object

You should declare a Java object as type JavaObject. For example:

```
Local JavaObject &MyJavaClass;
```

Note. Java objects can be declared as type Local only.

Scope of a Java Object

A Java object can e instantiated from PeopleCode only. This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

PeopleCode Java Built-in Functions

Chapter 1, PeopleCode Built-in Functions, CreateJavaArray , page 98

Chapter 1, PeopleCode Built-in Functions, CreateJavaObject , page 99

Chapter 1, PeopleCode Built-in Functions, GetJavaClass , page 252

CHAPTER 17

Message Class

This chapter provides an overview of the Message class and discusses the following topics:

- Application Message PeopleCode.
- Data type of a message object.
- Scope of a message object.
- Message object population.
- Error handling.
- Message class built-in functions.
- Message class methods.
- Message class properties.

Understanding Message Class

You can create the following types of messages in PeopleSoft Application Designer:

- *Structured* messages, which use record definitions to create a hierarchical structure for the data. These are generally used for data from applications, pages, components, and so on.
- *Unstructured* messages, which do not use record definitions. They are 'empty' structures, generally used for XML or SOAP messages.

Use the PeopleCode message class to create and access structured messages. Structured messages are self-describing messages that contain application data. They store the data passed between systems.

Message definitions are simply multi-level structures, consisting of records. Publishing a message basically stores the message in a set of relational tables, the application messaging queue.

Messages can be published (created) using an Application Engine program or component. Subscription PeopleCode programs subscribe to messages and process the data.

The Message class is built on top of the Rowset, Row, Record, and Field classes, so the PeopleCode written to populate or access a Message looks similar to PeopleCode written to populate or access the Component buffer.

See Also

PeopleSoft Integration Broker, Defining Message Channels and Messages, Defining a Message

PeopleCode Developer's Guide, Data Buffer Access

Messaging PeopleCode Events

The following PeopleCode events are associated with messages:

- OnRouteSend
- OnRouteReceive
- OnRequest
- Subscription

OnRouteSend and OnRouteReceive: These PeopleCode events are used with routing rules to determine where the messages are routed to and received from.

OnRequest: This PeopleCode event is used with synchronous messages, when a message has been received and requires a reply.

Subscription: This PeopleCode event fire when the message is subscribed to. Use this for your subscription processing for validating and loading message data.

See Also

PeopleSoft Integration Broker, Defining Message Channels and Messages, Accessing Message PeopleCode

Data Type of a Message Object

Messages are declared using the Message data type. For example,

```
Local Message &MSG;

Global Message &MyMessage;
```

Scope of a Message Object

A message object can only be instantiated from PeopleCode. This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

See Also

PeopleSoft Integration Broker, Defining Message Channels and Messages, Defining a Message

Message Object Population

After you've declared and instantiated your message object, you want to populate it with data. If your data is coming from Component buffers, populating your message is easy.

A message definition contains a hierarchy of records. A Component buffer contains a hierarchy of records. If you want to copy data from a Component buffer rowset to a message, the *structure* of the message and the component must be the same. That is, if you have a record at level 2 in your message and you want that data, you must have the same level 0 and level 1 records in your message as in your component.

For example, suppose your component had the following structure (that is, that PO_INFO and PO_LINE are at the same level, and PO_DETAIL is the child of PO_INFO):

```
PO_HEADER
   PO_LINE
   PO_INFO
      PO_DETAIL
```

To include the information in the PO_DETAIL record, you must have *at least* the following record structure in your message:

```
PO_HEADER
   PO_INFO
      PO_DETAIL
```

Any records that are in the page that aren't in the message (and vice-versa) are ignored.

After you get your message object, you can create a rowset from it. This rowset has the same structure as the message. If the message is empty, the rowset has no data. If the message has data, the rowset is automatically populated.

The following example is the simplest way of populating a message with data. This assumes that the structure of the message is the same as the structure of the page.

```
/* this gets all the data in the Component buffer */

&RS = GetLevel0();

/* this instantiates a message object */

&MSG = CreateMessage(MESSAGE.MY_MESSAGE);

   /* creates a rowset with the same structure as the message */

&MSG_RS = &MSG.GetRowset();

   /* this copies all the data from the page to the message */

&RS.CopyTo(&MSG_RS);

/* this publishes the message */
```

```
&MSG_RS.Publish();
```

A message rowset is the same as a Component buffer rowset, or any other rowset. It is composed of rows, records, and fields. Suppose you didn't want to get all the data from the Component buffer, but instead wanted to populate just a particular record in your message.

To access a record in a message rowset is the same as accessing a record in a Component buffer rowset. You must instantiate the rowset, then specify the row before you can access the record.

The following selects values into a record, then uses the record method **CopyFieldTo** to copy from the Component buffer record to the message record.

```
Local SQL &LN_SQL;
Local Message &STOCK_MSG;
Local Rowset &HDR_RS, &LN_RS;
Local Record &LN_REC, &ln_rec_msg;

&STOCK_MSG = CreateMessage(Message.STOCK_REQUEST);
&HDR_RS = &STOCK_MSG.GetRowset();
&LN_REC = CreateRecord(Record.DEMAND_INF_INV);

&LN_SQL = CreateSQL("Select * from PS_DEMAND_INF_INV where  BUSINESS_UNIT= :1 and⇒
 ORDER_NO = :2", &BUSINESS_UNIT, &ORDER_NO);

&J = 1;
While &LN_SQL.Fetch(&LN_REC)

   /* copy data into the Level 1 of &STOCK_MSG object */
   &LN_RS = &HDR_RS(&I).GetRowset(1);

   If &J > 1 Then
     &LN_RS.InsertRow(&J - 1);
   End-If;

   &ln_rec_msg = &LN_RS.GetRow(&J).GetRecord(Record.DEMAND_INF_INV);
   &LN_REC.CopyFieldsTo(&ln_rec_msg);
   &J = &J + 1;
End-While;
```

This section also discusses items to keep in mind when:

- Populating a rowset from a message.
- Publishing and subscribing to partial records.
- Subscribing to character fields.

Considerations When Populating a Rowset From a Message

Suppose your message had two rowsets, one at level 0, a second at level 1. In the message, only the level 0 rowset contains any data. When you use GetRowset to create a rowset for the entire message, the rowset at level 1 will contain an empty row, even if there isn't any data in it. (This is standard behavior for *all* rowsets.) However, you can use the IsChanged property on the record object to determine the status of the data.

The following subscription PeopleCode example traverse the Rowset. (Notice the use of **ChildCount**, **ActiveRowCount**, and **IsChanged**).

‘. . .’ is where application specific code would go.

```
&MSG_ROWSET = &MSG.GetRowset();
For &A0 = 1 To &MSG_ROWSET.ActiveRowCount

/**************************/
/* Process Level 1 Records */
/*-------------------------*/

   If &MSG_ROWSET(&A0).ChildCount > 0 Then
   For &B1 = 1 To &MSG_ROWSET(&A0).ChildCount

    &LEVEL1_ROWSET = &MSG_ROWSET(&A0).GetRowset(&B1);
    For &A1 = 1 To &LEVEL1_ROWSET.ActiveRowCount
    If &LEVEL1_ROWSET(&A1).GetRecord(1).IsChanged Then
     . . .

/**************************/
/* Process Level 2 Records */
/*-------------------------*/

     If &LEVEL1_ROWSET(&A1).ChildCount > 0 Then
     For &B2 = 1 To &LEVEL1_ROWSET(&A1).ChildCount
       &LEVEL2_ROWSET = &LEVEL1_ROWSET(&A1).GetRowset(&B2);
       For &A2 = 1 To &LEVEL2_ROWSET.ActiveRowCount
       If &LEVEL2_ROWSET(&A2).GetRecord(1).IsChanged Then
       .  .  .

/**************************/
/* Process Level 3 Records */
/*-------------------------*/

         If &LEVEL2_ROWSET(&A2).ChildCount > 0 Then
         For &B3 = 1 To &LEVEL1_ROWSET(&A2).ChildCount
         &LEVEL3_ROWSET = &LEVEL2_ROWSET(&A2).GetRowset(&B3);
         For &A3 = 1 To &LEVEL3_ROWSET.ActiveRowCount
         If &LEVEL3_ROWSET(&A3).GetRecord(1).IsChanged Then
            .  .  .

         End-If; /* A3 - IsChanged */
         End-For; /* A3 - Loop */
         End-For; /* B3 - Loop */
         End-If; /* A2 - ChildCount > 0 */

/*-------------------------------*/
/* End of Process Level 3 Records */
/*********************************/
```

```
                End-If; /* A2 - IsChanged */
          End-For; /* A2 - Loop */
        End-For; /* B2 - Loop */
        End-If; /* A1 - ChildCount > 0 */

/*--------------------------------*/
/* End of Process Level 2 Records */
/**********************************/

     End-If; /* A1 - IsChanged */
     End-For; /* A1 - Loop */
    End-For; /* B1 - Loop */
    End-If; /* A0 - ChildCount > 0 */

/*--------------------------------*/
/* End of Process Level 1 Records */
/**********************************/

End-For; /* A0 - Loop */
```

Considerations for Publishing and Subscribing to Partial Records

If you've selected to not publish all the fields in the message, you must be careful when inserting that data into a record.

Deselecting the Includecheck box in the message definition means that the field is *excluded* from the message definition.

- The field won't be included when generating an XML document (when publishing a message.)
- If the field is present in the XML (that is, it wasn't excluded from the published message) it is ignored by the subscribing system.

When you insert the data from the message into the database, you *must*set the values for the fields that aren't in the message. You can use the SetDefault Record class method to do this. You could also use a Component Interface based on the component the message was created from to leverage the component defaults.

See the Designing Component Interface Subscribe EIP for more information about using Component Interfaces in subscription.

See Also

Chapter 25, Record Class , SetDefault, page 1456

Considerations When Subscribing to Character Fields

If a message definition has character fields that are defined as uppercase, when the message is subscribed to, character data for those fields is automatically converted to uppercase.

Error Handling

In your subscription PeopleCode, you may want to validate the information received in the message. If you find that the data isn't valid, use Exit(1) to end your PeopleCode program. This sets the status of the message to ERROR, logs any error messages to the Application Message Queues, and automatically rolls back any database changes you may have made.

There are many ways to validate the data and capture the error messages:

- Use ExecuteEdits on the message object
- Use ExecuteEdits on the record object
- Use a Component Interface

See Designing Component Interface Subscribe EIP for more details.

Write your own validation PeopleCode in the Subscription program:

The following example validates the Business Unit of a message against an array of valid BU's:

```
For &I = 1 To &ROWSET.RowCount;

  &POSITION = &BUArray.Find(&ROWSET(&I).GetRecord(1).BUSINESS_UNIT.Value);
  If &POSITION = 0 Then
    &Status = "ERROR: BU not Found or not Active";
    &ROWSET(&I).BCT_ADJS_MSG_VW.BUSINESS_UNIT.IsEditError = True;
     &ROWSET(&I).BCT_ADJS_MSG_VW.BUSINESS_UNIT.MessageSetNumber = 11100;
    &ROWSET(&I).BCT_ADJS_MSG_VW.BUSINESS_UNIT.MessageNumber = 1230;
/* The error message 11100-1230 reads: This Business Unit is */
/* not a valid, open, Inventory Business Unit */

  End-If;

End-For;
```

In the calling PeopleCode, the program calls Exit(1) based on the value of &Status.

Note. All errors for Subscription PeopleCode get logged to the application message error table, *not* to the PSMessages collection (on the session object.)

See Also

Chapter 17, Message Class , ExecuteEdits, page 987

Chapter 25, Record Class , ExecuteEdits, page 1448

PeopleSoft Integration Broker, Sending and Receiving Messages, Processing Inbound Errors

Message Class Built-In Functions

Chapter 1, PeopleCode Built-in Functions, AddSystemPauseTimes , page 37

Chapter 1, PeopleCode Built-in Functions, CreateMessage , page 100

Chapter 1, PeopleCode Built-in Functions, DeleteSystemPauseTimes , page 143

Chapter 1, PeopleCode Built-in Functions, GetMessage , page 255

Chapter 1, PeopleCode Built-in Functions, GetSyncLogData , page 277

Chapter 1, PeopleCode Built-in Functions, PingNode , page 346

Chapter 1, PeopleCode Built-in Functions, ReturnToServer , page 377

Chapter 1, PeopleCode Built-in Functions, SetChannelStatus , page 401

Message Class Methods

In this section, we discuss the Message class methods. The methods are discussed in alphabetical order.

Clone

Syntax

```
Clone()
```

Description

The **Clone** method creates an identical copy of the message, copying the data tree of the message executing the method. The **Clone** function sets the following properties for the resulting message object, based on the values set for the message definition:

- Name
- ChannelName
- IsActive

Other properties are set when the message is published or subscribed to (PubID, SubName, and so on), or are dynamically generated at other times (Size, IsEditError, and so on.)

Clone creates a unique version of the message. If the original message changes, it is not reflected in the cloned object.

Parameters

None.

Returns

A message object.

Example

Clone could be used in a Hub and Spoke messaging environment. The Hub node uses this method during subscription processing to publish a copy of the messages it receives from the Spokes.

```
&Msg = GetMessage();

&Clone = &Msg.Clone();

&Clone.Publish();
```

The hub's publication routing rules are then used to determine which spokes receive the message.

See Also

Chapter 17, Message Class , CopyRowset, page 981, Chapter 1, PeopleCode Built-in Functions, CreateMessage , page 100, *PeopleCode Developer's Guide*, Understanding Objects and Classes in PeopleCode, Assigning Objects.

CopyRowset

Syntax

CopyRowset(*source_rowset* [, *record_list*]);

Where *record_list* is a list of record names in the form:
RECORD.*source_recname1*, RECORD.*target_recname1*
[, RECORD.*source_recname2*, RECORD.*target_recname2*]. . .

Description

The **CopyRowset** method copies *all* data from the source rowset to the *like-named* fields in the message object executing the method. This is an easy way to populate a message with data from a component.

Note. CopyRowset copies all the data, including rows that haven't been modified. If you want to copy only data that has changed in some way, use the CopyRowsetDelta method.

See Chapter 17, Message Class , CopyRowsetDelta, page 982.

When the record names in the message and component do not match exactly, use the optional *record_list* to specify the records to be copied from the component into the message.

When you use the **CopyRowset** method to copy the contents from the source rowset to the message object, you are creating a *unique* copy of the object. If you change the original rowset, the message object is not changed.

Note. You can execute CopyRowset against a message only once. After a message is populated, any other CopyRowsets are ignored. If you have to populate different levels of a message rowset separately, you can use the CreateRowset method to create a rowset that has the same structure as your message, populate the created rowset, then use CopyRowset to populate your message. You can also use the Rowset **CopyTo** method to populate a message rowset, then populate the PSCAMA record by hand.

Parameters

Parameter	Description
source_rowset	Specifies the name of the rowset to be copied into the message object.
record_list	Specifies specific source and target records to be copied into the message object.

Returns

None.

Example

The following example copies an entire component into a message object.

```
&Msg = GetMessage();
&Component_Rowset = GetLevel0();
&Msg.CopyRowset(&Component_Rowset);
```

The following example copies a header/line page rowset into a header/line message object, using the *record_list* because the record names don't match:

```
&Msg = GetMessage();
&Component_Rowset = GetLevel0();
&Msg.CopyRowset(&Component_Rowset, RECORD.PO_HDR_VW, RECORD.PO_HDR, RECORD.PO_LINE_⇒
VW, RECORD.PO_LINE);
```

See Also

Chapter 17, Message Class , CopyRowsetDelta, page 982, Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268, Chapter 27, Rowset Class , page 1479, Chapter 27, Rowset Class , Fill, page 1485.

CopyRowsetDelta

Syntax

CopyRowsetDelta(*source_rowset* [, *record_list*]);

Where *record_list* is a list of record names in the form:
[RECORD.*source_recname1*, RECORD.*target_recname1*
[, RECORD.*source_recname2*, RECORD.*target_recname2*]]. . .

Description

The **CopyRowsetDelta** method copies rows of data that have changed from the source rowset to the *like-named* records and *like-named* fields in the message object executing the method.

Note. CopyRowsetDelta copies all the like-named fields from the changed *row*into the message. It is *not* copying just the changed like-named fields.

When the record names in the message and component do not match exactly, use the optional *record_list* to specify the records to be copied from the component into the message. The specified target records must be records in your message, while the specified source records must be records in a rowset that exists in the data buffer and is populated.

This is an easy way to populate a message when the records in the message match the records in the component that the message is published from.

In addition, the **CopyRowsetDelta** method sets the AUDIT_ACTN field in the PSCAMA table for every row in the message. The subscription process can then use PSCAMA.AUDIT_ACTN to determine how to process every row that was published.

The set values match those used in audit trail processing, that is:

- A - Row inserted.
- D - Row deleted.
- C - Row changed (updated), but no key fields changed. The system copies the new value to the Message rowset.
- K - Row changed (updated), and at least one key field changed. The system copies the old value to the Message rowset and the new value (see "N").
- N - Row changed (updated), and at least one key field changed. The system copies the old value to the Message rowset and the new value (see "K").
- "" − blank value means that nothing on that row has changed. This is the default value, and is the value used to tag the parent rows of children that have changed.

Note. If a child row is inserted (or changed or deleted) CopyRowsetDelta also copies the parent row (and the parent row of that row, and so on, up to the top row) so the subscriber has a full picture of the transaction. A blank value is set in the AUDIT_ACTN field for these rows to let the subscriber know they don’t have to take action; the parent rows are there for reference only.

The Audit_Action values "A", "C", "D" are set when a record is added, changed, or deleted, respectively. In some cases such as effective-dated records, the user may change a key field value, such as Effective Date. In response to such an user action, two records are created, one with an Audit_Action value of "N", and the other with Audit_Action value "K". The "N" record has all the new values, while the "K" record retains the old values.

When you use the **CopyRowsetDelta** method to copy the contents from the source rowset to the message object, you are creating a *unique*copy of the object. If you change the original rowset, the message object is *not* be changed.

Note. You can execute CopyRowsetDelta against a message only once. After a message is populated, any other CopyRowsetDeltas are ignored. If you have to populate different levels of a message rowset separately, you can use the CreateRowset method to create a rowset that has the same structure as your message, populate the created rowset, then use CopyRowsetDelta to populate your message. You can also use the Rowset **CopyTo** method to populate a message rowset, then populate the PSCAMA record by hand.

Parameters

Parameter	Description
source_rowset	Specifies the name of the rowset to be copied into the message object.
record_list	Specifies source and target records to be copied into the message object. The target records must be records in your message, while the source records must be records in a rowset that exists in the data buffer *and*is populated.

Returns

None.

Example

The following example copies all the changed rows of data from a component into a message object.

```
&Component_Rowset = GetLevel0();

&Msg.CopyRowsetDelta(&Component_Rowset);
```

See Also

PeopleSoft Integration Broker, Sending and Receiving Messages, Understanding PSCAMA, *PeopleCode Developer's Guide*, Understanding Objects and Classes in PeopleCode, Assigning Objects, Chapter 17, Message Class , CopyRowset, page 981, Chapter 27, Rowset Class , page 1479, Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268.

CopyRowsetDeltaOriginal

Syntax

CopyRowsetDeltaOriginal(*source_rowset* [*, record_list*]);

Where *record_list* is a list of record names in the form:

```
[RECORD.source_recname1, RECORD.target_recname1
[, RECORD.source_recname2, RECORD.target_recname2]]. . .
```

Description

The **CopyRowsetDeltaOriginal** method copies rows of data that have changed from the source rowset to the *like-named* records and *like-named* fields in the message. It also copies the original value of the changed rows.

Note. CopyRowsetDeltaOriginal copies all the like-named fields from the changed and original *rows* into the message. It is *not* copying just the changed like-named fields.

When the record names in the message and component do not match exactly, use the optional *record_list* to specify the records to be copied from the component into the message. The specified target records must be records in your message, while the specified source records must be records in a rowset that exists in the data buffer and is populated.

The **CopyRowsetDeltaOriginal** method sets the AUDIT_ACTN field in the PSCAMA table for every row in the message. The subscription process can then use PSCAMA.AUDIT_ACTN to determine how to process every row that was published.

The set values match those used in audit trail processing, that is:

- A - Row inserted.
- D - Row deleted.
- C - Row changed (updated), but no key fields changed. The system copies the new value to the Message rowset and the original value (see "O" below).
- O - Prior value of changed row. The system copies the new value to the Message rowset and the original value (see "C").

- K - Row changed (updated), and at least one key field changed. The system copies the original value to the Message rowset and the new value (see "N").
- N - Row changed (updated), and at least one key field changed. The system copies the original value to the Message rowset and the new value (see "K").
- "" − blank value means that nothing on that row has changed. This is the default value, and is the value used to tag the parent rows of children that have changed.

Note. If a child row is inserted (or changed or deleted) CopyRowsetDeltaOriginalalso copies the parent row (and the parent row of that row, and so on, up to the top row) so the subscriber has a full picture of the transaction. A blank value is set in the AUDIT_ACTN field for these rows to let the subscriber know they don’t have to take action; the parent rows are there for reference only.

The Audit_Action values "A", "C", "D" are set when a record is added, changed or deleted, respectively. When a row is changed, two records are created, one with an Audit_Action of "C", the other with Audit_Action value of "O". The "C" record has all the new values, while the "O" record retains the original values. In some cases such as effective-dated records, a user may change a key field value, such as Effective Date. In response to such an user action, two records are created, one with an Audit_Action value of "N", and the other with Audit_Action value "K". The "N" record has all the new values, while the "K" record retains the original values. The "N"/"K" combination is also created if both a key change and a non-key change is made on the same row of data.

The following table details the output from CopyRowsetDeltaOriginal:

User Action	CopyRowsetDeltaOriginal Output	Comments
Change a key.	"N" and "K" rows created.	The "K" row contains the original Key value, the "N" row, the new.
Change a Key + change a Non-Key.	"N" and "K" rows created only (that is: no "C"/"O" rows in that case).	The Key and Non-Key Original Values are both contained in the "K" row.
Change a Non-Key.	"C" and "O" rows created.	Original Value stored in "O" row.
Add non-effective dated row.	"A" row created.	No additional rows created.
Add effective-dated row and only 1 original effective-dated row exists.	"A" and "O" rows created.	"O" row contains the original effective dated row.

User Action	CopyRowsetDeltaOriginal Output	Comments
Add effective-dated row and more than 1 original effected dated rows exist.	"A" and "O" rows created.	"O" row contains data from row that cursor was on when the new row was added. For example: If a rowset contains two rows (01/01/1980 and 02/02/2000), the current row is the 01/01/1980 row and the user adds a new row with today's date. The original values contain the data from the 01/01/1980. Likewise, if a rowset contains two rows (01/01/1980 and 02/02/2000), the current row is the 02/02/2000 row and the user adds a new row with today's date. Then the original values contain the data from the 02/02/2000.
Delete a row	"D" row created.	No additional rows created.

When you use the **CopyRowsetDeltaOriginal** method to copy the contents from the source rowset to the message object, you are creating a *unique*copy of the object. If you change the original rowset, the message object is *not* changed.

Note. You can execute CopyRowsetDeltaOriginal against a message only once. After a message is populated, any other CopyRowsetDeltaOriginal PeopleCode statements are ignored. If you have to populate different levels of a message rowset separately, you can use the CreateRowset method to create a rowset that has the same structure as your message, populate the created rowset, then use CopyRowsetDeltaOriginal to populate your message. You can also use the Rowset **CopyTo** method to populate a message rowset, then populate the PSCAMA record manually.

Parameters

Parameter	Description
source_rowset	Specifies the name of the rowset to be copied into the message object.
record_list	Specifies source and target records to be copied into the message object. The target records must be records in your message, while the source records must be records in a rowset that exists in the data buffer *and* is populated.

Returns

None.

Example

```
Function Update_Dir(&MSGName As string)
```

```
   &MSGName = "Message." | &MSGName;
   &MSG = CreateMessage(@(&MSGName));

   &PnlRowset = GetLevel0();
   &MSG.CopyRowsetDeltaOriginal(&PnlRowset);
   &MSG.Publish();

End-Function;
```

See Also

PeopleSoft Integration Broker, Sending and Receiving Messages, Understanding PSCAMA, *PeopleCode Developer's Guide*, Understanding Objects and Classes in PeopleCode, Assigning Objects, Chapter 17, Message Class , CopyRowset, page 981, Chapter 17, Message Class , CopyRowsetDelta, page 982, Chapter 27, Rowset Class , page 1479.

ExecuteEdits

Syntax

ExecuteEdits([*editlevels*]);

where *editlevels* is a list of values in the form:
editlevel1 [+ *editlevel2*] . . .]);

and where *editleveln* is one of the following system constants:

%Edit_DateRange

%Edit_OneZero

%Edit_PromptTable

%Edit_Required

%Edit_TranslateTable

%Edit_YesNo

Description

The **ExecuteEdits** method executes the standard system edits on every field for every record in the message executing the method. All edits are based on the record edits defined for the record that the message is based on. The types of edits performed depends on the *editlevel*. If no *editlevel* is specified, all system edits defined for the record definition are executed, that is:

- Reasonable Date Range (Is the date contained within plus or minus 30 days from the current date?)
- 1/0 (Do all 1/0 fields only contain 1 or 0?)
- Prompt Table (Is field data contained in the specified prompt table?)
- Required Field (Do all required fields contain data? For numeric or signed fields, it checks for NULL or 0 values.)
- Translate Table (Is field data contained in the translate table?)

• Yes/No (Do all yes/no fields only contain only yes or no data?)

If any of the edits fail, the status of the property **IsEditError** is set to True for the Message Object executing the method, and any Rowset, Row, Record, or Field Objects that could be instantiated from the same data as the Message Object. In addition, the Field class properties **MessageNumber** and **MessageSetNumber** are set to the number of the returned message and message set number of the returned message, for the field generating the error.

If you want to check the field values only for a particular record, you can use the **ExecuteEdits** method with a record object.

In addition, if you want to dynamically set the prompt tables used for a field (with the %EditTable function) you must use the **SetEditTable** method.

See Chapter 12, Field Class , EditError, page 854.

See Chapter 12, Field Class , MessageNumber, page 863.

See Chapter 12, Field Class , MessageSetNumber, page 863.

See Chapter 25, Record Class , ExecuteEdits, page 1448.

See Chapter 17, Message Class , SetEditTable, page 994.

Considerations for ExecuteEdits and SetEditTable

If an effective date is a key on the prompt table, and the record being edited doesn't contain an effective-dated field, the current datetime is used as the key value.

If a setID is a key on the prompt table, and the record being edited doesn't contain a setID field, the system looks for the setID on the other records in the current row first, then search parent rows.

Considerations for ExecuteEdits and Numeric Fields

A 0 (zero) may or may not be a valid value for a numeric field. ExecuteEdits processes numeric fields in different ways, depending on whether the field is required and a prompt table is used:

–If the numeric field is required and there is no prompt edit: 0 is considered valid, so numeric fields never fail the required test.

—If the numeric field is required and there is a prompt edit: Prompt edit is run and the value must be valid.

—If the numeric field is not required and there is a prompt edit: 0 is considered valid, so the prompt edit is skipped.

Parameters

Parameter	Description
editlevel	Specifies the standard system edits to be performed against every field on every record. If *editlevel* isn't specified, all system edits are performed. *editlevel* can be any of the following system constants: • %Edit_DateRange • %Edit_OneZero • %Edit_PromptTable • %Edit_Required • %Edit_TranslateTable • %Edit_YesNo

Returns

None.

Example

The following is an example of a call to execute Required Field and Prompt Table edits:

```
&MSG.ExecuteEdits(%Edit_Required + %Edit_PromptTable);
```

The following is an example showing how ExecuteEdits() could be used:

```
&MSG.ExecuteEdits();
If &MSG.IsEditError Then
  Exit(1);
End-If;
```

See Also

Chapter 17, Message Class , SetEditTable, page 994, Chapter 12, Field Class , EditError, page 854, Chapter 12, Field Class , MessageNumber, page 863, Chapter 12, Field Class , MessageSetNumber, page 863, Chapter 25, Record Class , IsEditError, page 1462, *PeopleSoft Integration Broker*, Sending and Receiving Messages, Processing Inbound Errors.

GenXMLString

Syntax

GenXMLString()

Description

The **GenXMLString** method creates an XML string based on the message after it's loaded from the message buffer.

Note. This method does *not* support unstructured messages.

Parameters

None.

Returns

An XML string.

Example

In the following example, the first thee lines of code creates the message object and copies the data from a Component buffer into the message object. The last line creates an XML string based on that message object.

```
&MSG = CreateMessage(MESSAGE.PO_INSERT);
&RS = GetLevel0();
&MSG.CopyRowset(&RS);
&XML_STRING = &MSG.GenXMLString();
```

See Also

Chapter 17, Message Class , LoadXMLString, page 992.

GetQueryString

Syntax

GetQueryString(*Parameter_Name*)

Description

Use the **GetQueryString** method to obtain the parameter values of a query string.

Parameters

Parameter	Description
Parameter_Name	Specify the parameter name you want to obtain.

Returns

A string containing the value of the parameter.

Example

```
Local Message &request;
Local String &emplid;
&request = GetMessage();
&emplid = &request.GetQueryString("EMPLID");
```

See Also

Chapter 17, Message Class , SetQueryString, page 995.

GetRowset

Syntax

GetRowset([*version*])

Description

The **GetRowset** method returns a standard PeopleCode level 0 rowset object for the specified message version. It creates an empty rowset for the specified message version if it doesn't already exist. If no message version is specified, the default message version is used.

When you use the **GetRowset** method, you are *not* creating a unique copy of the object. You are making only a copy of the reference. If you change the rowset, the original message is changed.

Version Considerations

If you specify a version, GetRowset won't return a rowset that is populated with the data for the specified version *unless* you call one of the following functions first, specifying the version:

- GetPubHeaderMessage
- GetPubMessage
- GetSubMessage

Parameters

Parameter	Description
version	An optional parameter, specifying an existing message version as a string. If *version* isn't specified, the default message version is used.

Returns

A Rowset object if successful.

Example

The following gets all the SET_CNTRL_REC rows related to the row on the page, then updates the field SETID with the value from the page. **GetRowset** is used to create a rowset based on the message structure, that is, an unpopulated rowset. Because the rowset is unpopulated, you can use the **Fill** method to populate with data from the page.

```
Local Message &MSG;
Local Rowset &MSG_ROWSET;

If FieldChanged(SETID) Then
  &MSG = CreateMessage(MESSAGE.SET_CNTRL_REC);
  &MSG_ROWSET = &MSG.GetRowset();
  &MSG_ROWSET.Fill("where SETCNTRLVALUE =:1 and REC_GROUP_ID =:2", SETCNTRLVALUE,⇒
 REC_GROUP_ID);

For &I = 1 To &MSG_ROWSET.ActiveRowCount
  &MSG_ROWSET.GetRow(&I).SET_CNTRL_REC.SETID.Value = SETID;
```

```
  &MSG_ROWSET.GetRow(&I).PSCAMA.AUDIT_ACTN.Value = "C";
End-For;

&MSG.Publish();

End-If;
```

See Also

Chapter 17, Message Class , Clone, page 980, Chapter 1, PeopleCode Built-in Functions, CreateMessage , page 100, Chapter 27, Rowset Class , page 1479, *PeopleCode Developer's Guide*, Understanding Objects and Classes in PeopleCode, Assigning Objects.

LoadXMLString

Syntax

LoadXMLString(*XMLstring*)

Description

The **LoadXMLString** function loads the message object executing the method with the XML string *XMLstring*.

Parameters

Parameter	Description
XMLString	Specify the XML string to be loaded into the message object.

Returns

None.

Example

```
&MSG = CreateMessage(MESSAGE.PO_HEADER);
&MSG.LoadXMLString(&XML_STRING);
```

See Also

Chapter 17, Message Class , GenXMLString, page 989.

Publish

Syntax

Publish([*Deferred_Mode*] [, *NodeName*])

Description

The **Publish** method publishes a message to the application message queue for the default message version. This method does not publish the message if the IsActive property for the message definition, set in PeopleSoft Application Designer, is False.

This method publishes a message asynchronously, that is, a reply message is not automatically generated. To publish a message synchronously, use the **SyncRequest** method.

Note. This method does *not* support unstructured messages.

The Publish method can be called from any PeopleCode event, but is generally called from an Application Engine PeopleCode step or from a component.

When publishing from a component, you should publish messages only from the SavePostChange event, either from record field or component PeopleCode. This way, if there's an error in your publish code, the data isn't committed to the database. Also, since SavePostChange is the last Save event fired, you avoid locking the database while the other save processing is happening.

If the message is successfully published, the PubID, and PubNodeName properties are set to the publication ID and publishing system node name.

Note. If you're publishing a message from within an Application Engine program, the message won't actually be published until the calling program performs a COMMIT.

Parameters

Parameter	Description
Deferred_Mode	Specify whether the message should be published immediately or at the end of the transaction. This parameter takes a Boolean value: False, publish the message immediately, or True, defer the publication. The default value is False.
NodeName	Specify the node to which the message should be published.

Returns

None.

Example

The following copies all the data from a page into a message and publishes it.

```
Local Message &MSG;
Local Rowset &ROWSET;

&MSG = CreateMessage(MESSAGE.MESSAGE_PO):
&ROWSET = GetLevel0();
&MSG.CopyRowset(&ROWSET);
&MSG.Publish;
```

The following is an example of putting the Incremental Publish PeopleCode on a Record at Level 1 (or 2, 3). If you just do the normal publishing PeopleCode, you get as many messages as there are records at that level (because the code fires for each record).

To make sure the code fires only once, use the following code.

```
/*NAMES.EMPLID.WORKFLOW */
Local Message &MSG;
Local Rowset &RS;

&LEVEL = CurrentLevelNumber();
```

```
&CURRENT_ROW = CurrentRowNumber(&LEVEL);
&TOT_ROW = ActiveRowCount(EMPLID);
/* Only Publish the message once for all records on */
/* this level */
If &CURRENT_ROW = &TOT_ROW Then
  &MSG = CreateMessage(MESSAGE.NAMES);
  &RS = GetRowset();
  &MSG.CopyRowsetDelta(&RS);
  &MSG.Publish();
End-If;
```

See Also

Chapter 17, Message Class , IsActive, page 1001, *PeopleSoft Integration Broker*, Defining Message Channels and Messages, Defining a Message.

SetEditTable

Syntax

SetEditTable(*%PromptField*, RECORD.*recordname*)

Description

The **SetEditTable** method works with the ExecuteEdits method. It is used to set the value of a field on a record that has its prompt table defined as %*PromptField* value. %PromptField values are used to dynamically change the prompt record for a field.

There are several steps to setting up a field so that you can dynamically change its prompt table.

To set up a field with a dynamic prompt table:

1. Define a field in the DERIVED record called *fieldname*.
2. In the record definition for the field that you want to have a dynamic prompt table, define the prompt table for the field as %*PromptField*, where *PromptField* is the name of the field that you created in the DERIVED record.
3. Use **SetEditTable** to dynamically set the prompt table.

%PromptField is the name of the field on the DERIVED work record. RECORD.*recordname* is the name of the record to be used as the prompt table.

```
&MSG.SetEditTable("%EDITTABLE1", Record.DEPARTMENT);
&MSG.SetEditTable("%EDITTABLE2", Record.JOB_ID);
&MSG.SetEditTable("%EDITTABLE3", Record.EMPL_DATA);
&MSG.ExecuteEdits();
```

Every field on a record that has the prompt table field %EDITTABLE1 will have the same prompt table, such as, DEPARTMENT.

Parameters

Parameter	Description
%PromptField	Specifies the name of the field in the DERIVED record that has been defined as the prompt table for a field in a record definition.
RECORD.*recordname*	Specifies the name of the record definition that contains the correct standard system edits to be performed for that field in the message.

Returns

None.

See Also

Chapter 17, Message Class , ExecuteEdits, page 987, Chapter 12, Field Class , EditError, page 854, Chapter 12, Field Class , MessageNumber, page 863, Chapter 12, Field Class , MessageSetNumber, page 863, Chapter 25, Record Class , IsEditError, page 1462, Chapter 25, Record Class , SetEditTable, page 1457, *PeopleSoft Application Designer*, Creating Record Definitions, Creating a New Record.

SetQueryString

Syntax

SetQueryString(*Parameter_Name, Value***)**

Description

Use the **SetQueryString** method to add a parameter value to the query string.

Parameters

Parameter	Description
Parameter_Name	Specify the name of the parameter that you want to add to the query string, as a string.
Value	Specify the value of the parameter you are adding to the query string.

Returns

None.

Example

```
Local Message &request;
&request = CreateMessage(Message.QE_SALES_ORDER);
. . .

. . .

&request.SetQueryString("BUYER", "12345");
```

See Also

Chapter 17, Message Class , GetQueryString, page 990.

SetStatus

Syntax

SetStatus(*Status*)

Description

Use the **SetStatus** method to set the status of a publication or subscription contract, much the same way you do in the message monitor.

You may want to use this method after an end-user has finished fixing any errors in the message data.

You can set the status of a contract to one of the following statuses:

- Cancel
- Done
- Error
- New

The method is available only when the message instance has one of the following statuses:

- Cancel
- Done
- Error
- New

Parameters

Parameter	Description
Status	Specify the status that you want to set the message to. Value are: • %Message_Error • %Message_New • %Message_Done • %Message_Started • %Message_Working • %Message_Retry • %Message_Timeout • %Message_Edited • %Message_Canceled

Returns

None.

See Also

PeopleSoft Integration Broker, Defining Message Channels and Messages, Defining a Message.

SyncRequest

Syntax

SyncRequest([*NodeName*])

Description

Use the **SyncRequest** method to send a synchronous message for the default message version.

Use the IsActive property to determine if a message is active or not before using this method. A message is specified as Active in PeopleSoft Application Designer.

This method sends a message synchronously, that is a reply message is returned as a result of this method. If you want to publish a message asynchronously, use the **Publish** method.

Note. This method does *not* support unstructured messages.

The SyncRequest method can be called from any PeopleCode event, but is generally called from an Application Engine PeopleCode step or from a component.

When sending a synchronous request from a component, you should send messages only from the SavePostChange event, either from record field or component PeopleCode. This way, if there's an error in your SyncRequest code, the data isn't committed to the database. Also, since SavePostChange is the last Save event fired, you avoid locking the database while the other save processing is happening.

If the message is successfully sent, a response message is returned. In addition, the GUID property is updated with a unique identifier.

Parameters

Parameter	Description
NodeName	Specify the name of the node that you want to send this message to.

Returns

A response message.

Example

```
Local Message &request, &response;
Local Rowset &sales_order;

&sales_order = GetLevel0();
&request = CreateMessage(Message.QE_SALES_ORDER);
&request.CopyRowsetDelta(&sales_order);
&response = &request.SyncRequest();
If (&response.ResponseStatus = 0) Then
   /* do processing */
End-If;
```

See Also

Chapter 17, Message Class , Publish, page 992.

Update

Syntax

Update([*versionlist*])

where *versionlist* is an arbitrary list of message versions in the following format:
`version1 [, version2] . . .`

Description

The **Update** method updates a message in the message queue with the specified message versions. If *versionlist* isn't specified, the default message version is used. This method is commonly used in the OnPublishTransform and OnSubscribeTransform PeopleCode events.

Note. This method does *not* support unstructured messages. The **Update** method can't be called from subscription PeopleCode.

Parameters

Parameter	Description
versionlist	Specify an optional comma-separated list of message versions. If *versionlist* isn't specified, the default message version is used.

Returns

None.

See Also

Chapter 17, Message Class , GetRowset, page 991, *PeopleSoft Integration Broker*, Defining Message Channels and Messages, Defining a Message.

Message Class Properties

In this section, we discuss the Message class properties. The properties are discussed in alphabetical order.

ChannelName

Description

This property references the name of the channel associated with the message definition. This property is set in PeopleSoft Application Designer, when the message is created. The message instance keys are a subset of the publication and subscription contract keys. This property is one of the message instance keys: the others are PubID and PubNodeName for asynchronous messages, GUID and PubNodeName for synchronous messages.

This property is valid for both synchronous and asynchronous messages.

This property is read-only.

Example

```
&CHANNEL = &MSG.ChannelName
```

See Also

PeopleSoft Integration Broker, Defining Message Channels and Messages, Defining a Message.

Cookies

Description

This property returns or sets the cookies associated with a message. You can accept a synchronous response message containing cookies, save those cookies in a global variable, and later return them to the target node in an outbound synchronous or asynchronous request message.

You can access this property only in an inbound synchronous response message or an outbound request message.

This property is read-write.

Example

The following is an example of receiving cookies:

```
Local Message &SalesRequest, &SalesResponse;
Local Rowset &SALES_ORDER;
Global string &SalesCookies;

&SALES_ORDER = GetLevel0();
&SalesRequest = CreateMessage(Message.SALES_ORDER_SYNC);
&SalesRequest.CopyRowsetDelta(&SALES_ORDER);

/* send the synchronous request; the return value is */
/* the response message object */
&SalesResponse = &SalesRequest.SyncRequest();

/* Retrieve cookies from the response message */
&SalesCookies = &SalesResponse.Cookies;
```

The following is an example of returning cookies:

```
Local Message &SalesRequest, &SalesResponse;
Local Rowset &SALES_ORDER;
Global string &SalesCookies;

&SALES_ORDER = GetLevel0();
&SalesRequest = CreateMessage(Message.SALES_ORDER_SYNC);
&SalesRequest.CopyRowsetDelta(&SALES_ORDER);

/* Insert the cookies in the request message */
&SalesRequest.Cookies = &SalesCookies;

/* Send the asynchronous request */
&SalesRequest.Publish();
```

DefaultMessageVersion

Description

This property returns the default version of the message as a string. This is the message version from the pub/sub contract, *not* the default message version of the message defined in the current system.

This property is valid for both synchronous and asynchronous messages.

This property is read-only.

Example

```
Local Message &MSG;
Local String &VER;

&MSG = GetMessage();
&VER = &MSG.DefaultMessageVersion
```

DoNotPubToNodeName

Description

Use this property to set the node name of a node that you do not want to publish this message to. For example, a single node may publish and subscribe to the same message. You can use this property to prevent the system from subscribing to the same message it publishes. This can help prevent circular processing where the original publisher eventually receives the same message.

This property is only valid for asynchronous messages.

This property is read-write.

Example

```
&MSG.DoNotPubToNodeName = &MSG.PubNodeName;
&MSG.Publish();
```

GUID

Description

This property returns a string representing a global unique identifier generated when the message was sent.

This property returns the unique identifier used with the message.

This property is valid only for synchronous messages.

This property is read-only.

Example

```
Local Message &request;
Local String &guid;

&request = GetMessage();
```

```
&guid = &request.GUID;
```

IsActive

Description

Indicates whether the message definition has been set to inactive in PeopleSoft Application Designer. This property is True if the message definition is active, False if it's been inactivated from PeopleSoft Application Designer. If you have a lot of PeopleCode associated with publishing a message, you might use this property to check if the message is active before you publish it.

This property is valid only for structured messages. If you want to find out whether an unstructured message is active or not, use the IsMessageActive built-in function.

This property is valid for both asynchronous and synchronous messages.

This property is read-only.

Example

```
&MSG = CreateMessage(MESSAGE.MY_MESSAGE)
If &MSG.IsActive Then
   /* do PeopleCode processing */
   &MSG.Publish();
Else
   /* do other processing */
End-if;
```

See Also

PeopleSoft Integration Broker, Defining Message Channels and Messages, Defining a Message, Chapter 1, PeopleCode Built-in Functions, IsMessageActive , page 307.

IsDelta

Description

This property indicates if *any* fields in a message populated from page data have been changed since they were first loaded into the component buffer.

This is generally used in the following scenario:

You want to publish a message only if the end-user has changed a value on a page. The problem is that if the end-user makes a change to a field in the level 3 rowset, the IsChanged property for the top level row remains *False*. If the rowset is large, and you don't want to iterate through it to find a single IsChanged, you can use this property to quickly determine if something has changed at a lower level and the message should be published.

Note. This property should be used only when the message and the component do *not* have the same structure (that is, the same records at the same levels). If the message and the component have the same structure use the CopyRowsetDelta method instead.

This property takes a Boolean value: True if there are changed fields in the message, False otherwise.

This property is valid for both asynchronous and synchronous messages.

This property is read-only.

Example

```
&Msg = CreateMessage(Message.MY_MESSAGE);
&Msg_Rowset = &Msg.GetRowset(); /* get first level in Msg RS */
&Msg_Detail = &Msg_Rowset(1).GetRowset(); /* get detail in Msg */
&Page_Rowset = GetRowset(); /* get page */
For &I = &Page_Rowset.Rowcount
   &Page_Detail = &Page_Rowset.GetRow(&I).GetRowset();
   &Msg_Detail.CopyRowset(&Page_Detail);
End-For;
/* Find if any data changed and should publish message */
If &Msg.IsDelta Then
   &Msg.Publish();
Else
   /* don't publish message and do other processing */
End-If;
```

See Also

Chapter 17, Message Class , CopyRowsetDelta, page 982.

IsEditError

Description

This property is True if an error has been found on any field in any record of the current message after executing the ExecuteEdits() method on either a message object or a record object. This property can be used with the Field Class properties MessageSetNumber and MessageNumber to find the error message set number and error message number.

You can use this property in Subscription PeopleCode with **Exit(1)**. Exit(1) automatically rolls back any changes that were made to the database, saves the message errors to the message queue, and marks the message status as ERROR.

This property is valid for both asynchronous and synchronous messages.

This property is read-only.

Example

```
&MSG = GetMessage();
&MSG.ExecuteEdits();
If &MSG.IsEditError Then
  Exit(1);
End-If;
```

See Also

Chapter 17, Message Class , ExecuteEdits, page 987, Chapter 12, Field Class , MessageNumber, page 863, Chapter 12, Field Class , MessageSetNumber, page 863, Chapter 1, PeopleCode Built-in Functions, Exit , page 176, *PeopleSoft Integration Broker*, Sending and Receiving Messages, Processing Inbound Errors.

IsEmpty

Description

This property indicates whether the transaction occurred without error, but returned an empty XML document. An empty response object contains only the data tags:

```
<data></data>
```

This property returns a Boolean value: True, the response message is empty, False, the response message is not empty.

This property is valid only for the returned response message from a synchronous request message.

This property is read-only.

IsLocal

Description

This property is True if the message object that was just instantiated was published locally. This property takes a Boolean value.

This property is valid for both asynchronous and synchronous messages.

This property is read-only.

Example

```
&MSG = GetMessage();
If &MSG.IsLocal Then /*  LOCAL NODE */
  /* do processing specific to local node */
Else
  /* do processing specific to remote nodes */
End-If;
```

MessageDetail

Description

This property specifies the message detail, as a string. The message detail can be used to further identify a message.

The only way to set the message detail is using PeopleCode. You cannot set the message detail using PeopleSoft Application Designer.

For synchronous messages, the message detail can be seen in the message monitor.

This property is valid with both asynchronous and synchronous messages.

This property is read-write.

Name

Description

This property returns the name of the message as a string.

This property is valid with both asynchronous and synchronous messages.

This property is read-only.

NRId

Description

This property returns the non-repudiation ID as a string. This property is populated with a unique string when the message is published.

This property is only valid with messages that use non-repudiation.

This property is read-only.

See Also

PeopleSoft Integration Broker, Configuring the Messaging System, Implementing Authentication and Nonrepudiation.

PubID

Description

This property refers to the publication identifier, which is a number. It's generated for asynchronous messages when the message is published, and is sequential, based on the number of messages published for that channel. The message instance keys are a subset of the publication and subscription contract keys. This property is one of the message instance keys: the others are ChannelName and PubNodeName.

This property is valid only with asynchronous messages.

This property is read-only.

Example

```
&MSG.Publish();
&PUBID = &MSG.PubID;
```

PubNodeName

Description

This property refers to a string that contains the name of the publishing node. It is generated by the system when the message is published. The message instance keys are a subset of the publication and subscription contract keys. This property is one of the message instance keys: the others are PubID and ChannelName.

This property is valid for both asynchronous and synchronous messages.

For a synchronous message, this property returns the name of the requesting node.

This property is read-only.

Example

```
&MSG = GetMessage();
&PUBNODE = &MSG.PubNodeName;
```

ResponseStatus

Description

This property is valid only for the returned response message from a synchronous request message.

This property indicates whether a response was successful or not.

Note. This is *not* the same as SetStatus, which is used only with the message monitor.

This property returns a numeric value: 0, the response was successful, any other number indicates an error.

This property is read-only.

Size

Description

This is the approximate size of the uncompressed XML data generated when the message is published. The size is approximate for the following reasons:

- The maximum size of fields is used except for the case of long text and image fields.
- The active row count is used, without regard to whether the rows have been changed. In a CopyRowsetDelta, rows that are not changed, are not published.
- It doesn't include the size of the XML tags.

This property can be used with the system property %MaxMessageSize in a batch Application Engine program to prevent the application from publishing a message that is too large.

This property is valid for both asynchronous and synchronous messages.

This property is read-only.

Example

```
If &MSG.Size > %MaxMessageSize Then
  &MSG.Publish();
Else
  /*Move more data into the message object */
End-If;
```

See Also

Chapter 3, System Variables , %MaxMessageSize , page 570, *PeopleSoft Integration Broker*, Sending and Receiving Messages.

SubName

Description

This property refers to a string that contains the name of the subscription process currently processing the message. It is available only when GetMessage() is used in a subscription PeopleCode program.

This property is only valid for asynchronous messages.

This property is read-only.

Example

```
&MSG = GetMessage();
&SUBNAME = &MSG.SubName
```

SubscriptionProcessId

Description

This property returns a string containing the subscription process identifier. This is a unique sequential number.

This property is populated only if the “Generate Subscription Process Instance” option is turned on in the Subscription Process definition.

The subscription process identifier corresponds to the subscription process instance, *not* the message instance. If there are multiple subscription processes for the same message each subscription process will have a different, unique ID.

If the subscription process terminated abnormally, its process instance is lost, and a new one is generated on the next retry (that is, there will be a gap in the sequence.)

This property is valid only for asynchronous messages.

This property is read-only.

Considerations for Using SubscriptionProcessId

Consider the following when using SubscriptionProcessId:

- Using the "Run PeopleCode" option from Message Subscription does *not*generate the number. It is generated only if the Application Server is running the PeopleCode.
- If for some reason you have to rerun the subscription, a new number is assigned.
- Because generating the number is optional, your program *must* contain code to support the option being turned off. Here is a code example from ITEM_SYNC:

```
If All(&MSG.SubscriptionProcessId) Then
   &EIP_CTL_ID = Generate_EIP_CTL_ID(4, &MSG.SubscriptionProcessId);
Else
    &EIP_CTL_ID = Generate_EIP_CTL_ID(1, "Random");
End-If;
```

- If the flag is off, this property is blank. In this case, you may want to adopt a standard for generating a random number, as shown in the previous example.

See Also

PeopleSoft Integration Broker, Sending and Receiving Messages.

CHAPTER 18

Mobile Classes

This chapter provides an overview of the Mobile classes and discusses the following topics:

- Mobile classes overview
- Mobile terms
- Mobile hierarchy
- HTML areas used in mobile
- Debugging in mobile
- Error handling
- Data types for mobile
- Scope of a mobile object
- Mobile reference

See Also

PeopleTools Mobile Agent PeopleBook

PeopleCode Developer's Guide, Understanding Mobile Pages and Classes

Chapter 32, SyncServer Class, page 1633

Mobile Classes Overview

PeopleSoft Mobile Agent extends the functionality of PeopleSoft Internet Architecture (PIA) to disconnected mobile devices, allowing users to continue working with their PeopleSoft applications on a laptop computer or personal digital assistant (PDA) while disconnected from the Internet or a local network.

When you create mobile pages for display on a mobile device (a laptop or PDA), you create the following in PeopleSoft Application Designer:

- synchronizable Component Interfaces.
- mobile pages based on your synchronizable Component Interfaces.

In addition, you may create PeopleCode programs that manipulate the data. You use the Mobile Classes in a mobile application to work with the data on a mobile device.

This document provides an overview of using the Mobile Classes, including:

- Understanding the Mobile Device and the Component Interface
- Understanding differences between Mobile and PIA

See Also

Chapter 32, SyncServer Class, page 1633

Understanding the Mobile Device and the Component Interface

When the synchronization server and the mobile device are synchronized, the following items are downloaded to the mobile device:

- instances of synchronizable Component Interfaces.
- mobile pages.
- associated mobile PeopleCode programs.

When a user of Mobile Agent synchronizes with the synchronization server the first time, the relavant data for the synchronizable Component Interface is loaded to the mobile device.

During subsequent synchronizations, the data on the mobile device is synchronized with the database on the synchronization server.

This section discusses the items that are downloaded to the mobile device. For more information about the synchronization process, see the Synchronization Server documentation.

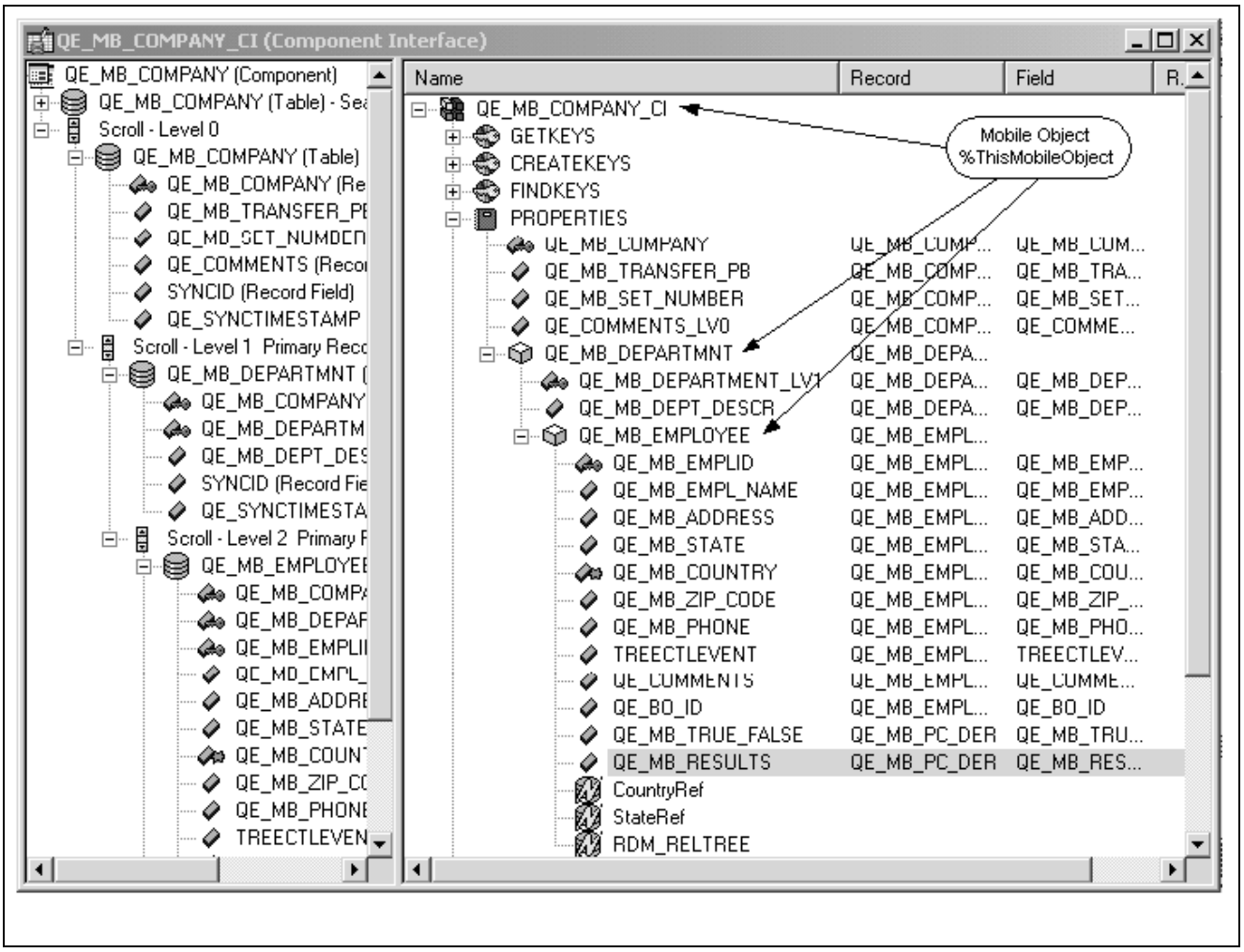
See Chapter 32, SyncServer Class, page 1633.

Synchronizable Component Interface

A synchronizable Component Interface is a Component Interface that is created specifically for use with a mobile device. This means that the Component Interface definition has been specified as synchronizable on the Component Interface properties page.

See *PeopleTools Mobile Agent PeopleBook*.

The following shows a synchronizable Component Interface.



Synchronizable Component Interface

Level 0 of the synchronizable component interface, as well as the data in the collections, are known as *mobile objects,* and are described in the following sections.

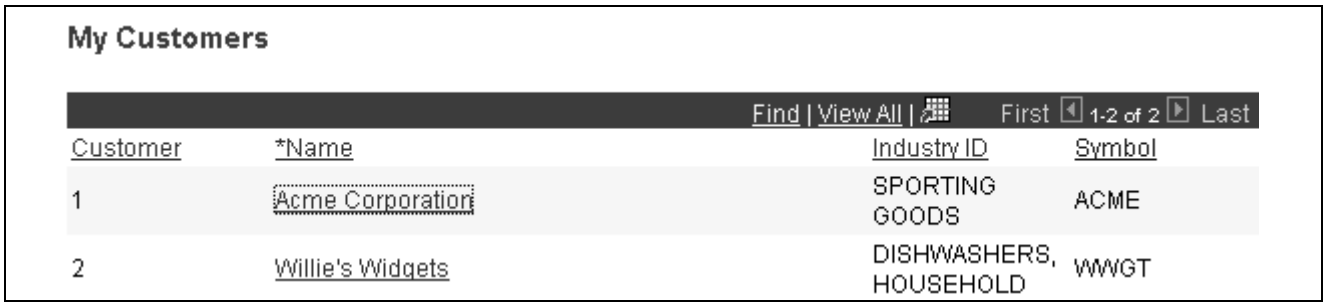
Mobile Page

The Mobile Page enables a mobile device user to change data in the synchronizable Component Interface that underlies the Mobile Page. The changes to the Component Interface data are sent back to the PeopleSoft database when the mobile device is synchronized.

Mobile pages have *list views* and *detail views.*

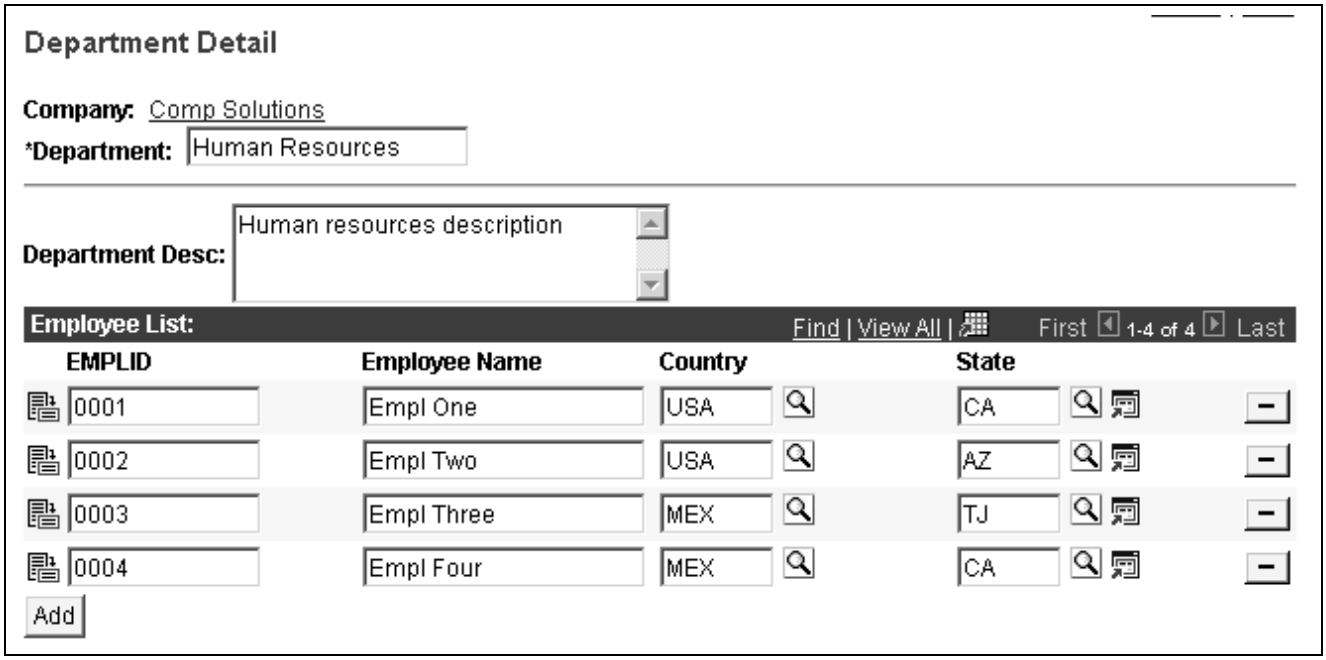
- List view pages are created from the top level of a component interface (the level 0), or a collection in a component interface. A list view can also be thought of as a search page.
- Detail view pages are created from the content properties (fields) in a mobile page. Where and how a property displays on a page is determined by the content properties page for that property (field).

The following is the list view for a mobile page, generated by the level 0 of the synchronizable component interface.



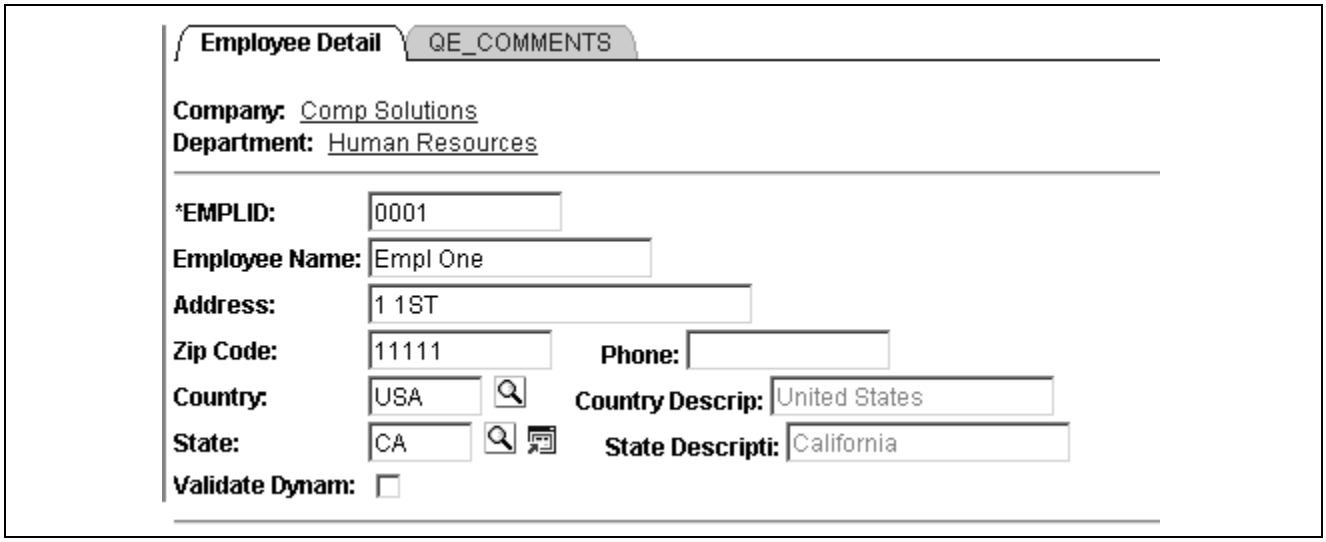
Mobile Page: Level 0 List View

The following is a list view generated by a collection in a synchronizable component interface.



Mobile page: Collection List View

The following is the detail view generated by the properties (fields) of the synchronizable component interface.



Mobile page: Detail View

Associated Mobile PeopleCode Programs

Access PeopleCode associated with a Component Interface by right clicking on an open synchronizable Component Interface in PeopleSoft Application Designer and selecting ViewPeopleCode.

The following is a PeopleCode program associated with the OnChange Mobile Component Interface event on a synchronizable Component Interface. The OnChange event is associated with a property on the page. The PeopleCode program uses a *mobile object* to access the QE_MB_EMPLID property on level 0 on the synchronizable Component Interface that has been downloaded to the mobile device. The mobile object represents the PIA equivalent of a row of data, and can be used at any level in the data hierarchy, level 0, level 1, level 2, and so on. It is what is displayed in the Detail View of a Mobile Page.

When a mobile event occurs on a mobile object, such as a mobile object property being changed (the OnChange event) or a mobile object being loaded into the working set (the OnInit event), the mobile object is then available in the event's PeopleCode program using the %ThisMobileObject system variable.

```
If %ThisMobileObject.IsNew() Then

/* do further check before save */

End-if;
```

Considerations Using the Mobile Object

A mobile object is the means in PeopleCode to access the synchronizable Component Interface. Mobile objects exist in relationship to each other in a hierarchy, which has a Component Buffer-like structure. A mobile object is just one row. A Data Collection is a collection of rows, each row being a mobile object.

The mobile object hierarchy has the equivalent of rowsets as data collections, rows as mobile objects, records and fields as properties. The topmost mobile object for a Component Interface is the equivalent of level 0 on the underlying Component Interface. A Component Interface can also contain levels, which map to mobile object data collections. (These levels, in PIA, map to scrolls on a page, or in the Component Buffer, to rowsets.)

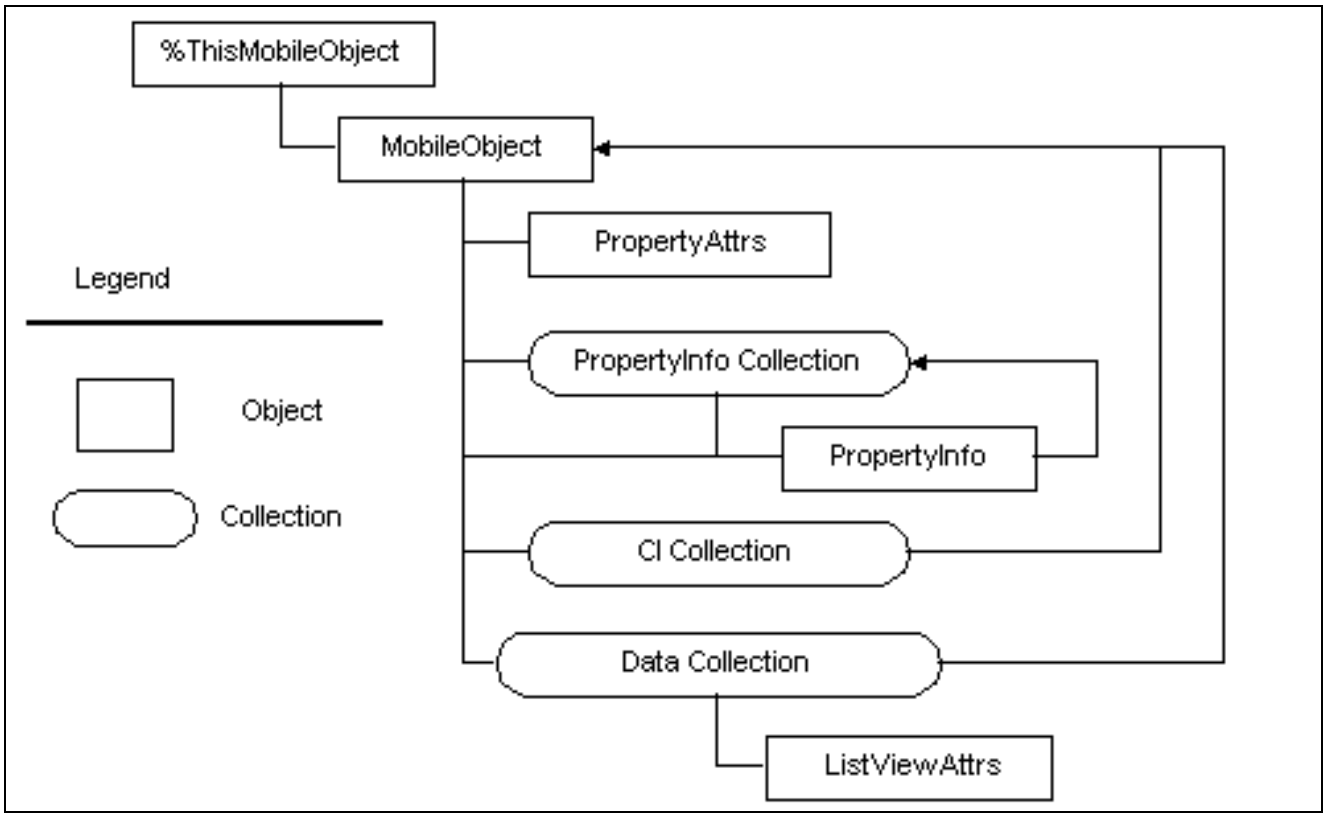
In the example in this section, the heirarchy of the mobile objects is:

```
QE_MB_COMPANY
    QE_MB_DEPARTMNT
        QE_MB_EMPLOYEE
```

- QE_MB_COMPANY — level 0.
- QE_MB_DEPARTMNT — level 1. This is a collection of mobile data objects.
- QE_MB_EMPLOYEE — level 2. This is a collection of mobile data objects.

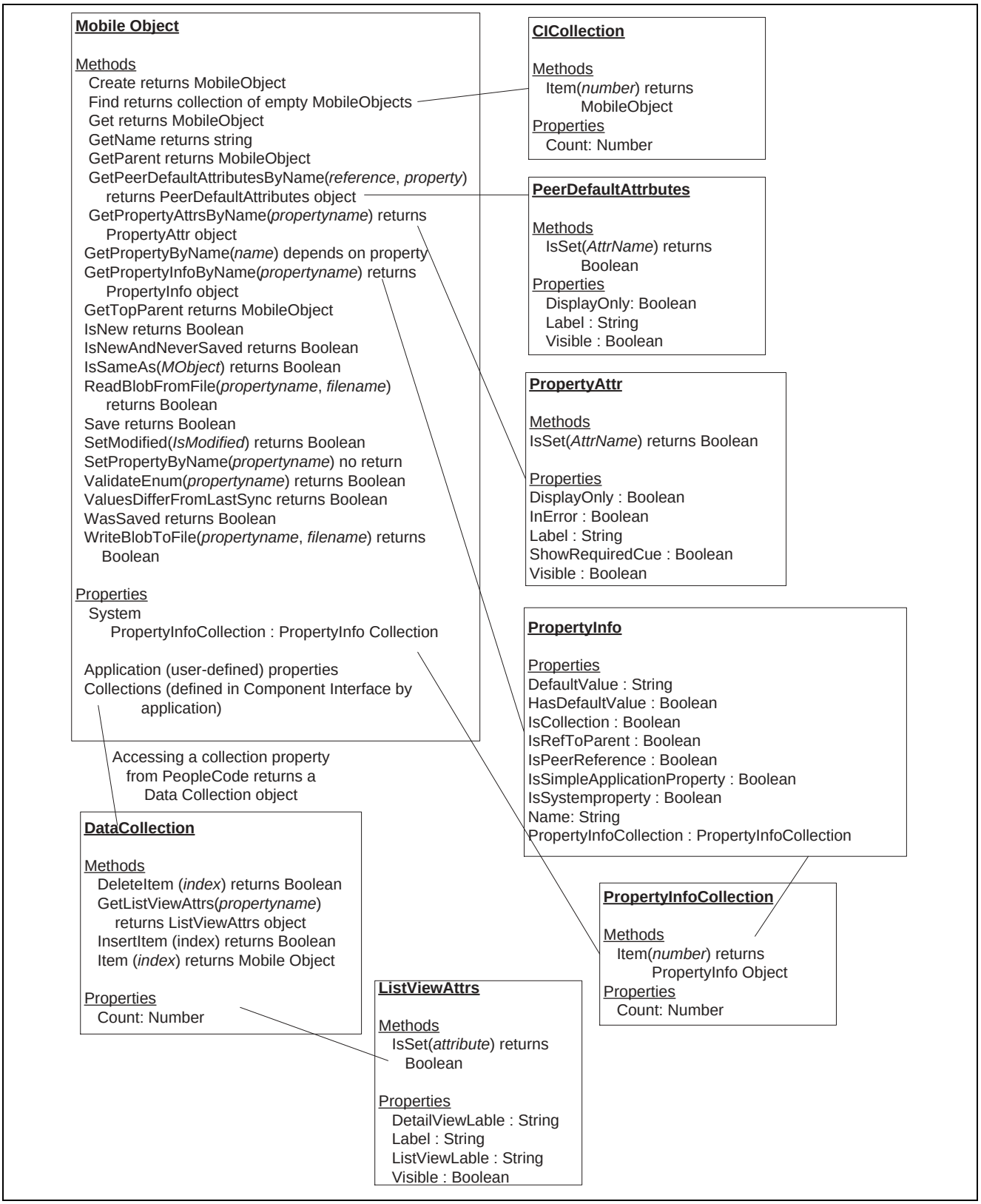
Mobile Class Hierarchy

Mobile objects exist in relationship to each other in a hierarchy. The following diagram shows the different classes, and how one class can be returned from a method or a property in another class.



Mobile classes hierarchy

The following presents the mobile object hierarchy in a class diagram:



Mobile classes hierarchy — class diagram

The following is a list of the various classes, with a brief description of each.

Mobile Class	Description
Mobile	Represents the PIA equivalent of a row of data, and can be used at any level in the data hierarchy, level 0, level 1, level 2, and so on. It is what is displayed in the Detail View of a mobile page.
CI collection	Contains a collection of empty mobile objects.
Data collection	Used to access a level on a mobile page, such as level 1, level 2, and so on.
ListViewAttrs	Used to access the display attributes for a list view.
PeerDefaultAttributes	Used to access the default display attributes for a peer reference.
PropertyAttrs	Used to access the display attrubutes for a property.
PropertyInfo	Used to access the attributes of a property (field) on a mobile page.
PropertyInfo collection	Contains a collection of PropertyInfo objects. Used when a property on a mobile page is actually a collection (such as a reference to level 1, level 2, and so on.)

HTML Areas Used in Mobile

You can use an HTML area on a mobile page, in either a list view or a detail view.

To set up an HTML area in mobile:

1. Add a long text field to a synchronizable Component Interface.
2. Create a mobile page from the synchronizable Component Interface.

   Remember, if you change the underlying Component Interface, any mobile page based on that Component Interface is *not* automatically changed.
3. On the associated mobile page definition, check the *Render as HTML Area* checkbox for the property on the property page.
4. Decide where the HTML area is to display.

   HTML areas are either displayed inline on the same page or on a separate tab. If you want to display the HTML area inline, check the *Display Inline on Same Page* checkbox for the property on the propery page.
5. In the OnInit event, populate the HTML area using PeopleCode.

You can either use HTML in your PeopleCode, or you can use the GetHTMLText function to access an existing HTML definition.

If you want to use images in your HTML, use the %IMAGE(*Imagename*) meta-HTML function.

6. Add the HTML definition to a project.
7. Specify the project that you added the HTML definition to as the project to be synchronized with the Component Interface.

   On the Synchronization tab of the Component Interface properties, you can specify a project to be synchronized with the Component Interface using the Related Content Project field. This project should contain all the HTML definitions, as well as all the images that you want distributed to the mobile device.

The following is an example of a test in PeopleCode:

```
/* HTML Area for Company  */
%ThisMobileObject.QE_COMMENTS_LV0 = "<H1>This is HTML for Company Page,
testing Mobile HTML Area</H1><H2>This is more HTML, testing Mobile HTML Area</H2>";
```

See Also

PeopleCode Developer's Guide, Using Methods and Built-In Functions, Using the HTML Area

Chapter 4, Meta-HTML, %Image , page 587

Debugging in Mobile

The psmobile.ini file contains debug settings for mobile PeopleCode.

By setting the values in the psmobile.ini file, you can use the PeopleCode debugger with your mobile application.

In addition, you can also trace some of the operations in the application. All trace information is written to a file named `psmobile.log`, located in the directory defined by `%temp%`.

Note. Be aware that tracing logs can grow to a large size. Only specify tracing when necessary, and only trace critical operations.

For a laptop, you can edit the psmobile.ini file on the mobile device. This file is either in the C:\Windows or C:\WinNT directory.

For a PDA, you must copy the psmobile.ini file to the laptop or desktop using ActiveSync, edit it there, then copy it back.

The following is an example of the psmobile.ini file.

```
[PSMOBILE]
Language=ENG
Port=8080
HomeDirectory=C:\Program Files\PeopleSoft\PeopleSoft Mobile Agent
MaxCachedStatements=100
MaxCachedObjects=100
debug=1
```

```
[SYNC]
SyncGateway=http://fsampson040502.corp.peoplesoft.com:80/SyncServer
SyncLogVerbose=0
[PCDEBUG]
DebugEnabled=0
DebugDatabase=Your Database Name
DebugOprid=Your OPRID
DebugIPAddress=nnn.nnn.nnn.nnn
DebugPort=9500
[Trace]
;-------------------------------------------------------------
; PeopleCode Tracing Bitfield
;
; Bit       Type of tracing
; ---       ---------------
; 1         - Trace instructions
; 2         - List the program
; 4         - Show assignments to variables
; 8         - Show fetched values
; 16        - Show stack
; 64        - Trace start of programs
; 128       - Trace external function calls
; 256       - Trace internal function calls
; 512       - Show parameter values
; 1024      - Show function return value
; 2048      - Trace each statement in program
; Dynamic change allowed for TracePC and TracePCMask
TracePC=0
```

Debug Settings

The following are the debug settings you can set in the psmobile.ini file.

DebugEnabled Specify whether debugging is enabled or not. Use a 0 to turn off debug mode, a 1 to turn on debug mode.

DebugDatabase Specify the actual database (not the domain) that application server uses with the PeopleTools PeopleCode debugger (this should be the database that is used for synchronizing).

DebugOperid Specify the userId you use to log onto the PeopleTools PeopleCode debugger.

DebugIPAddress Specify the IP address or machine name of the application server.

DebugPort Specify the port number that the application server (PSDBGSRV) has been configured to use.

Trace Settings

The following are the trace settings you can specify in the psmobile.ini file. You can specify more than one trace by adding the numbers together.

Note. Be aware that tracing logs can grow to a large size. Only specify tracing when necessary, and only trace critical operations.

0	Turn all tracing off.
1	Trace all instructions.
2	List the program being traced.
4	Show all assignments to all variables.
8	Show all fetched values.
16	Show stack.
64	Trace start of programs.
128	Trace external function calls.
256	Trace internal function calls.
512	Show all parameter values.
1024	Show function return value.
2048	Trace each statement in program.

Error Handling

For initial error handling, check the return values of methods and functions as they are used. When you want to check for errors depends on your application. However, if you check for errors after every assignment, you may see a performance degradation.

Mobile does not use collections of errors or messages (PSMESSAGES), as is the case with PIA. However, you can specify a message set and number for use with a mobile page and with mobile properties. This is done in Application Designer, to ensure that the correct messages get downloaded to the mobile device.

Once the messages have been distributed to the mobile device, you can access them using PeopleCode the same way you would access any message in the message catalog, using the following functions:

- MessageBox
- MsgGet
- MsgGetText
- MsgGetExplainText

In the following example, the program checks for an invalid value for a field, and issues a message if the value is incorrect. This code would be in an OnEdit event associated with a property.

```
&Value = %ThisMobileObject.EmailID;

If Not(Find("@", &Value)) Then
     /* invalid email address probably */
     MessageBox(0, "", 525, 2, "Message Not Found");

     &PropAttrs = %ThisMobileProperty.GetPropertyAttrsByName("EmailID");
     &PropAttrs.InError = True

End-If;
```

See Also

PeopleTools Mobile Agent PeopleBook, Using Synchronizable Component Interfaces, Specifying Messages

Data Types for Mobile

All mobile objects are declared as type ApiObject. For example:

```
Local ApiObject &MyMobileCollection;

Local ApiObject &MyMobileObject = %ThisMobileObject;
```

Note. Mobile objects can be declared only as Local.

Scope of Mobile Objects

A mobile object can be instantiated from PeopleCode. In addition, it can only be instantiated within the mobile events.

Variables in Mobile PeopleCode

Local variables are the only type of PeopleCode variables supported in mobile.

Mobile Functions

Chapter 1, PeopleCode Built-in Functions, GenerateMobileTree, page 215

Chapter 1, PeopleCode Built-in Functions, TransferMobilePage, page 467

Session Class Method

The following is the Session class method used with Mobile.

See Also

Chapter 29, Session Class , page 1559

GetCompIntfc

Syntax

GetCompIntfc([**COMPINTFC**.]*ComponentName*)

Description

The **GetCompIntfc** method returns a reference to an empty mobile object. Mobile objects are the means of accessing the synchronizable Component Interface in PeopleCode, so this is similar to creating an empty Component Interface in PIA.*ComponentName*, when used by itself, takes a string value. If you use COMPINTFC.*ComponentName* it is not a string value; it is a constant that automatically is renamed in your code if the Component Interface definition is ever renamed.

Parameters

Parameter	Description
ComponentName	Specify the name of the synchronizable Component Interface. You can either use *ComponentName* by itself or prefaced with the keyword **COMPINTFC**.

Returns

This method returns an empty mobile object if successful, or Null on error (such as *ComponentName* specifies a nonexistent synchronizable Component Interface).

Example

```
Local ApiObject &SaveCustomer, &Customer;

&SaveCustomer = %ThisMobileObject.MyFavoriteCustomer;

&Customer = %Session.GetCompIntfc(COMPINTFC.CUSTOMER);
&Customer.Name="Acme Applicances";

if (&Customer.Get() <> true) then;
   if (&Customer.Create() <> true) then;
      ErrorHandler();
   Else
      &Customer.Save();
   End-If;
End-if;
```

```
%ThisMobileObject.MyFavoriteCustomer = &Customer;
```

Mobile Class

The *mobile object* represents the PIA equivalent of a row of data. It is what is displayed in the Detail View of a Mobile Page.

The mobile object is accessed using the %ThisMobileObject system variable.

Mobile Class Methods

The following are the mobile class methods, in alphabetical order.

Note. With the mobile class methods, and unlike the Component Interface class methods, all properties that have been set are used in the search. This provides more flexibility, and does not require setting up specific Get, Create, or Find keys.

Create

Syntax

```
Create()
```

Description

The **Create** method associates the mobile object with a new, open mobile object. It is available only on an empty mobile object that is returned by **GetCompIntfc**.

Parameters

None.

Returns

A Boolean value: True if object was successfully created, False otherwise.

Example

```
Local ApiObject &SaveCustomer, &Customer;

&SaveCustomer = %ThisMobileObject.MyFavoriteCustomer;

&Customer = %Session.GetCompIntfc(COMPINTFC.CUSTOMER);
&Customer.Name="Acme Applicances";

if (&Customer.Get() <> true) then;
   if (&Customer.Create() <> true) then;
```

```
      ErrorHandler();
    Else
      &Customer.Save();
    End-If;
End-if;

%ThisMobileObject.MyFavoriteCustomer = &Customer;
```

See Also

Chapter 18, Mobile Classes , GetCompIntfc, page 1021

Find

Syntax

Find()

Description

The **Find** method returns a reference to a CI collection that is populated with a collection of empty mobile objects. The properties that the application developer sets within the empty mobile object are used as the Find keys. This method is available only on an empty mobile object that is returned by **GetCompIntfc**. Using it on a non-empty mobile object causes an error.

This collection may have zero elements in it if no items matching the key values you set were found. If you set no keys, all of the mobile objects of the type specified by GetCompIntfc are returned.

While the PIA Component Interface Find method limits the number of returned instances to 300, the mobile **Find** method has no limit.

The **Find** method returns only the topmost parent mobile objects. You do not have to set values for all the key values. After you have a collection, you can use the CI Collection method **Item** to return the empty mobile object, and then use **Get** on the mobile object to instantiate it.

Note. Wildcards are not implemented for this release.

Parameters

None.

Returns

A CICollection collection of mobile objects. You must use the **Get** method on these mobile objects before you can use them.

Example

```
/* Error checking left out for clarity */

&MyCI = %Session.GetCompIntfc(CompIntfc.QE_MB_COMPANY_CI);
&MyCI.QE_MB_COMPANY = "Acme Appliances";
&MyCIFindCollection = &MyCI.Find();
&Acme = &MyCIFindCollection.Item(1);
```

```
&Acme.Get();
```

Get

Syntax

Get()

Description

The **Get** method associates a mobile object with the user-defined properties that have been explicitly set on the object.

Parameters

None.

Returns

A Boolean value: True if mobile object was successfully retrieved, False otherwise.

Example

```
Local ApiObject &SaveCustomer, &Customer;

&SaveCustomer = %ThisMobileObject.MyFavoriteCustomer;

&Customer = %Session.GetCompIntfc(COMPINTFC.CUSTOMER);
&Customer.Name="Acme Applicances";

if (&Customer.Get() <> true) then;
    if (&Customer.Create() <> true) then;
      ErrorHandler();
  Else
      &Customer.Save();
  End-If;
End-if;

%ThisMobileObject.MyFavoriteCustomer = &Customer;
```

GetName

Syntax

GetName()

Description

The **GetName** method returns the class name of the mobile object.

Parameters

None.

Returns

The class name of the mobile object as a string. This is the name of the synchronizable Component Interface, or one of the levels.

Example

```
&GetName = &qe_mb_obj_setup.GetName();

/*   Write the name of the Object to the Results field */
&qe_mb_obj.QE_MB_RESULTS = "The name of this Mobile Object is: " | &GetName;
```

GetParent

Syntax

```
GetParent()
```

Description

The **GetParent** method returns a mobile object that is the parent of the Mobile object that is executing the method. If you are at the top level (level 0), it returns itself.

Parameters

None.

Returns

The parent mobile object.

Example

```
/* This loop will continue getting parent level, until level 0 is reached.  */
&obj = %ThisMobileObject;
Repeat
    &obj = &obj.getparent();
    /* Do some processing */
until (&obj.IsSameAs(%ThisMobileObject.gettopparent()));
```

GetPeerDefaultAttributesByName

Syntax

```
GetPeerDefaultAttributesByName(Reference, Property)
```

Description

When a peer reference does not have a value associated with it, the display attributes can only be accessed using the PeerDefaultAttributes class. After a peer reference has values, it can access using the ListViewAttrs class.

Use the **GetPeerDefaultAttributesByName** method to access the default peer reference attributes for the specified property on the peer reference.

Parameters

Parameter	Description
Reference	Specify the name of a peer reference, as a string.
Property	Specify a property on the peer reference, as a string.

Returns

A reference to a PeerDefaultAttributes class.

Example

```
%ThisMobileObject.GetPeerDefaultAttributesByName("CI_CUSTOMER_MB", "ADDRESS").⇒
Visible = True;
```

See Also

Chapter 18, Mobile Classes , PeerDefaultAttributes Class, page 1056

GetPropertyAttrsByName

Syntax

GetPropertyAttrsByName(*propertyname*)

Description

The **GetPropertyAttrsByName** method returns a reference to a PropertyAttrs object, used to modify display attributes of properties.

Parameters

Parameter	Description
propertyname	The name of the property.

None.

Returns

A PropertyAttrs object, used to modify display attributes of properties.

Example

```
%ThisMobileObject.GetPropertyAttrsByName("QE_MB_SET_DATE").Visible = False;
```

See Also

Chapter 18, Mobile Classes , PropertyAttrs Class, page 1052

GetPropertyByName

Syntax

GetPropertyByName(*Name*)

Description

The **GetPropertyByName** method returns the value of a property. Generally this function is used only in applications that cannot get the names of the mobile object user-defined properties until runtime.

Parameters

Parameter	Description
Name	Specify the name of the property you want to access. The PeopleCode program terminates if this property does not exist.

Returns

The value of the property specified by *Name*. For a collection, it returns a reference to the collection.

Example

```
&GetPropertyByName_Char = &qe_mb_obj_setup.GetPropertyByName(&CharacterProperty⇒
Name);
/*Write the results of the GetPropertyByName method to the results Field   */

&qe_mb_obj.QE_MB_RESULTS = "The value of the Character Field: "
| &GetPropertyByName_Char | "
```

GetPropertyInfoByName

Syntax

GetPropertyInfoByName(*propertyname*)

Description

The **GetPropertyInfoByName** method returns a reference to a PropertyInfo object.

Parameters

Parameter	Description
propertyname	The name of the property.

Returns

A PropertyInfo object.

Example

```
&pinfo = %ThisMobileObject.GetPropertyInfoByName("QE_MB_SET_DATE");
if &pinfo.HasDefaultValue = True then
   %ThisMobileObject.QE_MB_SET_DATE = &pinfo.DefaultValue;
end-if;
```

See Also

Chapter 18, Mobile Classes , PropertyInfo Class, page 1038

GetTopParent

Syntax

```
GetTopParent()
```

Description

The **GetTopParent** method returns a mobile object that is the topmost parent of the Mobile object.

Parameters

None.

Returns

A reference to the topmost parent mobile object as a mobile object. If the current mobile object is the topmost object, this method return a reference to the current mobile object.

Example

```
/* Check if the created Mobile Object is at Level 0  */
If (&qe_mb_obj_setup.IsSameAs(&qe_mb_obj_setup.GetTopParent())) Then

   /* Set the Parent Level variable to 0.
   Results for this get set below the looping statement  */
   &Parent_Level = 0;

End-If;
```

IsNew

Syntax

IsNew()

Description

The **IsNew** method indicates if this object has never been synchronized.

Parameters

None.

Returns

A Boolean value: True if this object was created on the device and never synchronized, False if this object has been synchronized.

Example

```
&IsNew = &qe_mb_obj_setup.IsNew();

/*Write the results of the IsNew to the results Field   */

&qe_mb_obj.QE_MB_RESULTS = "The following is " | &IsNew | ": The Mobile Object is⇒
 new, and has never been synced.";
```

IsNewAndNeverSaved

Syntax

IsNewAndNeverSaved()

Description

The **IsNewAndNeverSaved** method indicates if this object was created on the device and has not yet been saved.

Parameters

None.

Returns

A Boolean value: True if this object was created on the device and has not yet been saved, False otherwise.

Example

```
&IsNewAndNeverSaved = &qe_mb_obj_setup.IsNewAndNeverSaved();

/*Write the results of the IsNewAndNeverSaved method to the results Field   */

&qe_mb_obj.QE_MB_RESULTS = "The following is " | &IsNewAndNeverSaved | ": This⇒
 Mobile Object is new, and has never been saved. "
```

IsSameAs

Syntax

IsSameAs(*MObject*)

Description

The **IsSameAs** method determines if one mobile object is the same as another. This is useful when you are trying to determine if the %ThisMobileObject is the topmost mobile object.

Parameters

Parameter	Description
MObject	Specify the mobile object to which this mobile object is being compared.

Returns

A Boolean value: True if this mobile object is the same as the mobile object, False otherwise.

Example

```
/*Check if the created Mobile Object is at Level 0  */
If (&qe_mb_obj_setup.IsSameAs(&qe_mb_obj_setup.GetTopParent())) Then

/* Set the Parent Level variable to 0. Results for this get set below the looping⇒
 statement  */
&Parent_Level = 0;

EndIf;
```

ReadBlobFromFile

Syntax

ReadBlobFromFile(*blobpropertyname, filename*)

Description

The **ReadBlobFromFile** method reads a file on your mobile device and writes it into the attachment property. The attachment property is a reference property on the synchronizable Component Interface; its purpose is to contain attachment data.

When the mobile user synchronizes from the mobile page, the attachment file data contained in the property is sent to the synchronization server. The size of the attachment on the synchronization server is updated to reflect the new contents.

If you use this method, your synchronized Component Interface must use an attachment: the synchronization server must use its attachment events, and must run the attachment method PutAttachment.

See Chapter 32, SyncServer Class, Using Attachment Events, page 1640.

Working With Relative Paths

You can specify either an absolute or a relative path for *filename* .

If you specify a relative path, the path that is appended is the working directory for the PSMobile.exe file. The exact path depends on the installation of Mobile on the device.

For example, if you specified `\files\MyFile.ext` for *filename*, the actual path might be something like `c:\Program Files\PeopleSoft\Mobile\files\Myfile.ext`.

Parameters

Parameter	Description
blobpropertyname	A string specifying the name of the reference property used for the attachment.
FileName	This parameter is a string specifying the path and filename of the file that you use as an attachment.

Returns

This is a Boolean value: True if the attachment was read successfully; False otherwise.

Example

```
%ThisMobileObject.ReadBlobFromFile("Attachment", %ThisMobileObject.⇒
QE_MB_READ_WRITE | %ThisMobileObject.ATTACHUSERFILE);
```

Save

Syntax

Save()

Description

The **Save** method saves the entire mobile object hierarchy in which the mobile object is contained.

You do not generally call the **Save** method on %ThisMobileObject; that saving is done when the user clicks on the Save button.

Using this method does not save associated peer references; you must use the **Save** method on each peer references separately.

The standard PeopleSoft save business rules (that is, any PeopleCode programs associated with SaveEdit, SavePreChange, and so on) are executed after you execute this method.

If you instantiate another mobile object with PeopleCode (using Get, Create) these objects are not automatically saved when the end user clicks the Save button. You must use the Save method with these instances separately.

Using the Save method on developer-instantiated mobile objects does not guarantee they will be saved. The save does not take effect until the end user clicks the Save button.

If you make changes to a developer-instantiated mobile object, the end user is not automatically prompted to save when trying to leave the page because it is not part of the current component.

If a developer uses the Save method with a developer-instantiated component interface, and if the end user did not make any other changes to data in the current component interface, when the user leaves the current unit of work, they are *not* prompted to save their data. To ensure that the data actually gets committed to the database, the developer can force a prompt to occur. This can be done using the following code:

```
%ThisMobileObject.SetModified(True);
```

See Also

PeopleCode Developer's Guide, Understanding Mobile Pages and Classes, Mobile Component Transaction Model

Parameters

None.

Returns

A Boolean value: True if component was successfully saved, False otherwise.

Example

```
Local ApiObject &SaveCustomer, &Customer;
&SaveCustomer = %ThisMobileObject.MyFavoriteCustomer;
&Customer = %Session.GetCompIntfc(COMPINTFC.CUSTOMER);
&Customer.Name="Acme Applicances";
if (&Customer.Get() <> true) then;
    if (&Customer.Create() <> true) then;
      ErrorHandler();
  Else
      &Customer.Save();
  End-If;
End-if;
%ThisMobileObject.MyFavoriteCustomer = &Customer;
```

SetModified

Syntax

SetModified(*IsModified*)

Description

The **SetModified** method forces the mobile object to have the specified modified value of True or False.

Note. Not only does using this method change the overall modified state of the object, but when changing the modified state to False it also clears the modified flag on any properties that had been modified.

Parameters

Parameter	Description
IsModified	Specify whether this mobile object has been modified. This parameter takes a Boolean value: True, the mobile object has been modified, False, the object has not been modified.

Returns

The current setting of this mobile object *before* being modified by this method: True for modified before this setting, False for not modified before this setting.

Example

```
/*   Get the value of the TrueFalse check box, to see how to set the modified⇒
 setting.  */
If &qe_mb_obj.QE_MB_TRUE_FALSE = True Then
&True_or_False = True;
&SetModified = &qe_mb_obj_setup.SetModified(True);
Else
&True_or_False = False;
&SetModified = &qe_mb_obj_setup.SetModified( False);
End-If;
```

SetPropertyByName

Syntax

SetPropertyByName(*propertyname*, *value*)

Description

The **SetPropertyByName** method sets the value of a property. Generally this function is used only in applications that cannot set the names of the component interface properties until runtime.

Parameters

Parameter	Description
propertyname	The name of the property.
value	This is the value to which the property is set.

Returns

None. The PeopleCode program terminates if there is an error.

Example

```
%ThisMobileObject.SetPropertyByName("QE_MB_NUMBER", 77777);
```

ValidateEnum

Syntax

ValidateEnum(*propertyname*)

Description

The **ValidateEnum** method verifies if the value of a property is a valid enumeration value. Enumeration values are originally set in PeopleSoft Application Designer, when the original records for the synchronizable Component Interface are designed.

Parameters

Parameter	Description
propertyname	Specify the name of the property as a string.

Returns

A Boolean value: True if the value is a valid dynamic enumeration, False otherwise.

Example

```
/* Validate if the value entered into Results field is a valid dynamic enumeration⇒
 */
&ob = %ThisMobileObject;

&Validate_DE = &ob.ValidateEnum(&ob.QE_MB_RESULTS);

/*Enter Results into Results Field  */
&ob.QE_MB_RESULTS = "The following statement is " | &Validate_DE | ": The Property⇒
 " | &ob.QE_MB_RESULTS | " is a valid Dynamic Enumeration.";
```

ValuesDifferFromLastSync

Syntax

ValuesDifferFromLastSync()

Description

The **ValuesDifferFromLastSync** method indicates if at least one property has a value that differs from its value since the last synchronization.

If this object has been created on the mobile device, that is, when there is no previous sync value, the first saved values are used as the comparison values for this property.

This is a true member by member comparison of data values, not modified flags. The SetModified method has no effect on this method.

Parameters

None.

Returns

A Boolean value: True if at least one property in this object has a value that differs from its original value, False otherwise.

Example

```
&ValuesDifferFromLastSync = &qe_mb_obj_setup.ValuesDifferFromLastSync();

/*Write the results of the ValuesDifferFromLast Sync method to the results Field  ⇒
 */

&qe_mb_obj.QE_MB_RESULTS = "The following is " | &ValuesDifferFromLastSync | ": At⇒
 least one property has a value that differs from the values last synced."
```

WasSaved

Syntax

```
WasSaved()
```

Description

The **WasSaved** method indicates if this object has been saved since it was created or last synchronized.

Parameters

None.

Returns

A Boolean value: True if this object has been saved since it was created or last synchronized, False otherwise.

Example

```
&WasSaved = &qe_mb_obj_setup.WasSaved();

/*Write the results of the WasSaved method to the results Field   */

&qe_mb_obj.QE_MB_RESULTS = "The following is " | &WasSaved | ": This Mobile Object⇒
 has been saved since it was last synced, or created. "
```

WriteBlobToFile

Syntax

```
WriteBlobToFile(blobpropertyname, filename)
```

Description

The **WriteBlobToFile** method reads the attachment property and writes it into a file on your mobile device. The attachment property is a reference property on the synchronizable Component Interface; its purpose is to contain attachment data.

When the mobile user synchronizes from the mobile page, this attachment file data contained in the property is received from the synchronization server.

If you use this method, your synchronized Component Interface must use an attachment: the synchronization server must use its attachment events, and must run the attachment method GetAttachment.

See Chapter 32, SyncServer Class, Using Attachment Events, page 1640.

Working With Relative Paths

You can specify either an absolute or a relative path for *filename* .

If you specify a relative path, the path that is appended is the working directory for the PSMobile.exe file. The exact path depends on the installation of Mobile on the device.

For example, if you specified `\files\MyFile.ext` for *filename*, the actual path might be something like `c:\Program Files\PeopleSoft\Mobile\files\Myfile.ext`.

Parameters

Parameter	Description
blobpropertyname	A string specifying the name of the reference property that is used for the attachment.
FileName	A string specifying the path and filename of the attachment file. This parameter takes a string value.

Returns

This is a Boolean value: True if the attachment was written successfully; False otherwise.

Example

```
%ThisMobileObject.WriteBlobToFile("Attachment", %ThisMobileObject.QE_MB_READ_WRITE⇒
 | %ThisMobileObject.ATTACHUSERFILE);
```

Mobile Class Property

The following is the Mobile Class property.

Note. Normally, properties for mobile objects are the equivalent of the user-define properties on the sychronizable Component Interface. The properties described here should not be used for user-defined properties when one designs a synchronizable Component Interface in the PeopleSoft Application Designer.

PropertyInfoCollection

Description

The **PropertyInfoCollection** property returns a reference to a collection of PropertyInfo objects that describe the shape of the mobile object.

This property is read-only.

See Also

Chapter 18, Mobile Classes , PropertyInfo Class, page 1038

PropertyInfoCollection

The PropertyInfoCollection objects contains a collection of PropertyInfo objects.

A PropertyInfoCollection is returned from the following:

- a mobile object
- a PropertyInfo object

See Also

Chapter 18, Mobile Classes , PropertyInfoCollection, page 1037

Chapter 18, Mobile Classes , PropertyInfoCollection, page 1042

PropertyInfoCollection Method

The following is the method for the PropertyInfoCollection property.

Item

Syntax

Item(*number*)

Description

The Item method returns a PropertyInfo object that exists at the *number* position in the PropertyInfoCollection.

Parameters

Parameter	Description
number	A position in the PropertyInfoCollection collection. The first position is 1.

Returns

A reference to the PropertyInfo object that exists at the *number* position in the PropertyInfoCollection collection. If the specified item is a collection, the returned PropertyInfo object is invalid and is set to NULL.

Example

```
/* Copy all like properties from &Item to &TargetItem */
for &i = 1 to &Item.PropertyInfoCollection.Count
   if &Item.PropertyInfoCollection.Item(&i).IsSimpleApplicationProperty then
     try &TargetItem.SetPropertyByName(&Item.PropertyInfoCollection.Item(&i).Name,
        &Item.GetPropertyByName(&Item.PropertyInfoCollection.Item(&i).Name))
     catch Exception &xException;

     end-try;
   end-if;
end-for;
```

PropertyInfoCollection Property

The following is the property for the PropertyInfoCollection property.

Count

Description

This property returns the number of PropertyInfo objects in the PropertyInfoCollection .

This property is read-only.

Example

```
/* Copy all like properties from &Item to &TargetItem */
for &i = 1 to &Item.PropertyInfoCollection.Count
   if &Item.PropertyInfoCollection.Item(&i).IsSimpleApplicationProperty then
    try &TargetItem.SetPropertyByName(&Item.PropertyInfoCollection.Item(&i).Name,
       &Item.GetPropertyByName(&Item.PropertyInfoCollection.Item(&i).Name))
       catch Exception &xException;
     end-try;
   end-if;
end-for;
```

PropertyInfo Class

The PropertyInfo class objects are returned by the following:

- the Item method of a PropertyInfo collection

- the GetPropertyInfoByName of a mobile object

PropertyInfo Class Properties

The following are the properties for the PropertyInfo object.

DefaultValue

Description

This property returns the default value for this mobile object property.

This property is read-only.

Note. Definitional defaults are applied only when an Object is Created, just before the OnInit PeopleCode is run.

Example

```
&pinfo = %ThisMobileObject.GetPropertyInfoByName("QE_MB_SET_DATE");
if &pinfo.HasDefaultValue = True then
   %ThisMobileObject.QE_MB_SET_DATE = &pinfo.DefaultValue;
end-if;
```

HasDefaultValue

Description

This property returns a Boolean value: True if this mobile object property has a default value; False otherwise.

This property is read-only.

Note. Definitional defaults are applied only when an object is created, just before the OnInit PeopleCode is run.

Example

```
&pinfo = %ThisMobileObject.GetPropertyAttrsByName("Text");
if &pinfo.HasDefaultValue = True then;
   %ThisMobileObject.Text = &pinfo.DefaultValue;
end-if;
```

IsCollection

Description

This property returns True if the property that the PropertyInfo object is describing is a Data Collection, False otherwise. If IsCollection is False, the object represents a mobile object user-defined or system property.

This property is read-only.

Example

```
/* Use the PropertyInfoCollection Property of the mobile object class, and use the⇒
 Item number field
to determine the item method from the PropertyInfoCollection class. */
&PropertyInfo = &qe_mb_obj_setup.PropertyInfoCollection.item(&qe_mb_obj.QE_MB_PC_⇒
ITEM);

/*Write the results to the results field  */
&qe_mb_obj.QE_MB_RESULTS = "The following is " | &PropertyInfo.IsCollecton | ⇒
": This Property represents a collection";
```

IsRefToParent

Description

This property is True if this is a reference to this object's parent, False otherwise. A property that is RefToParent is also a System Property.

This property is read-only.

Example

```
/* Use the PropertyInfoCollection Property of the mobile object class, and use the⇒
 Item number
field to determine the item method from the PropertyInfoCollection class. */
&PropertyInfo = &qe_mb_obj_setup.PropertyInfoCollection.item(&qe_mb_obj.QE_MB_PC_⇒
ITEM);

/* Write the results to the results field  */
&qe_mb_obj.QE_MB_RESULTS = "The following is " | &PropertyInfo.IsRefToParent | ⇒
": This is a reference to this Object' Parent. "
```

IsPeerReference

Description

This property is True if this is a peer reference, False otherwise.

This property is read-only.

IsSimpleApplicationProperty

Description

This property is True is this is a simple, non-reference application-defined property; False otherwise. In other words, if True, it is not a system property, collection, nor peer reference; it is a property that was defined on the synchronized Component Interface in PeopleSoft Application Designer.

This property is read-only.

Example

```
/* Copy all like properties from &Item to &TargetItem */
for &i = 1 to &Item.PropertyInfoCollection.Count
   if &Item.PropertyInfoCollection.Item(&i).IsSimpleApplicationProperty then
      try &TargetItem.SetPropertyByName(&Item.PropertyInfoCollection.Item⇒
(&i).Name,
        &Item.GetPropertyByName(&Item.PropertyInfoCollection.Item(&i).Name))
      catch Exception &xException;

      end-try;
   end-if;
end-for;
```

IsSystemProperty

Description

This property is True is this is a System Property; False otherwise. System properties include the OID and VERSION properties, and references to Parents.

This property is read-only.

Example

```
/* Use the PropertyInfoCollection Property of the mobile object class, and use the⇒
 Item number
field to determine the item method from the PropertyInfoCollection class. */
&PropertyInfo = &qe_mb_obj_setup.PropertyInfoCollection.item(&qe_mb_obj.QE_MB_PC_⇒
ITEM);

/*Write the results to the results field  */
&qe_mb_obj.QE_MB_RESULTS = "The following is " | &PropertyInfo.IsSystemProperty |⇒
 ": This is a system property. "
```

Name

Description

The name of the property.

This property is read-only.

Example

```
/* Use the PropertyInfoCollection Property of the mobile object class, and use the⇒
 Item number
field to determine the item method from the PropertyInfoCollection class. */
&PropertyInfo = &qe_mb_obj_setup.PropertyInfoCollection.item(&qe_mb_obj.QE_MB_PC_⇒
ITEM);

/*Write the results to the results field  */
```

```
&qe_mb_obj.QE_MB_RESULTS = "The name of the accessed Property is: " | &Property⇒
Info.Name;
```

PropertyInfoCollection

Description

This property returns a reference to another PropertyInfoCollection collection if the object executing the property is a collection.

This property is read-only.

Example

```
/* Use the PropertyInfoCollection Property of the mobile object class, and use the⇒
 Item number field to determine the item method from the PropertyInfoCollection⇒
 class. */
&PropertyInfo = &qe_mb_obj_setup.PropertyInfoCollection.item(&qe_mb_obj.QE_MB_PC_⇒
ITEM);

/*Write the results to the results field. Since this property returns a CompIntf⇒
PropertyInfoCollect Object, use the Count porperty of the PropertyInfoCollection⇒
 class, to test this.*/
&qe_mb_obj.QE_MB_RESULTS = "The property count of the CompIntfProperyInfo⇒
Collection returned by this property is: " | &PropertyInfo.PropertyInfoCollection⇒
.count;
```

CI Collection Class

A CI Collection is returned by the mobile object Find method. A CI Collection is a collection of empty mobile objects.

See Also

Chapter 18, Mobile Classes , Find, page 1023

CI Collection Method

The following is the CI Collection method.

Item

Syntax

Item(*index*)

Description

The **Item** method returns a Mobile Component Interface object. Once you've returned a reference, to instantiate it, use the **Get** method on the returned object.

Parameters

Parameter	Description
index	A number value: the index number for the Component Interface to fetch. Indexes start at 1.

Returns

An empty mobile object that was fetched. You must use the Get method to get the data for that object.

Example

```
/* Error checking left out for clarity */
&MyCI = %Session.GetCompIntfc(CompIntfc.QE_MB_COMPANY_CI);
&MyCI.QE_MB_COMPANY = "Acme Appliances";
&MyCIFindCollection = &MyCI.Find();
&Acme = &MyCIFindCollection.Item(1);
&Acme.Get();
```

CI Collection Properties

The following is the CI Collection property.

Count

Description

The number of Mobile Component Interface objects in the CI Collection collection. This property is read-only.

Example

```
/* Error checking left out for clarity */
&MyCI = %Session.GetCompIntfc(CompIntfc.QE_MB_COMPANY_CI);
&total_in_collection = &MyCI.count;
```

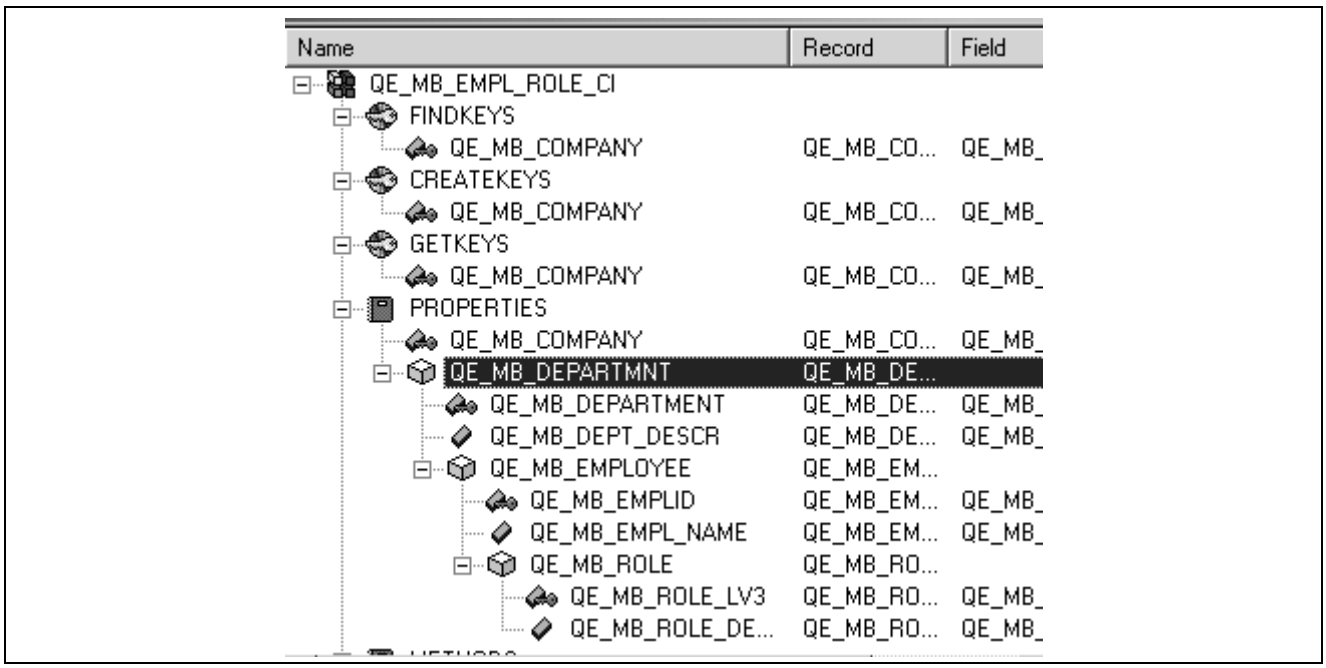
DataCollection Class

A Data Collection object is created when you have a Mobile Page that is based upon a synchronizable Component Interface that contains levels. You do not directly instantiate the DataCollection object.

The synchronizable Component Interface upon which the following Mobile Page is based has the following levels:

- level 1: QE_MB_DEPARTMNT
- level 2: QE_MB_EMPLOYEE
- level 3: QE_MB_ROLE

Use the DataCollection methods and property to access the mobile objects that map to the rows on these levels.



Component Interface Data Collection Levels

Data Collection Methods

Following are the methods for the DataCollection class.

GetListViewAttrs

Syntax

GetListViewAttrs(*propertyname*)

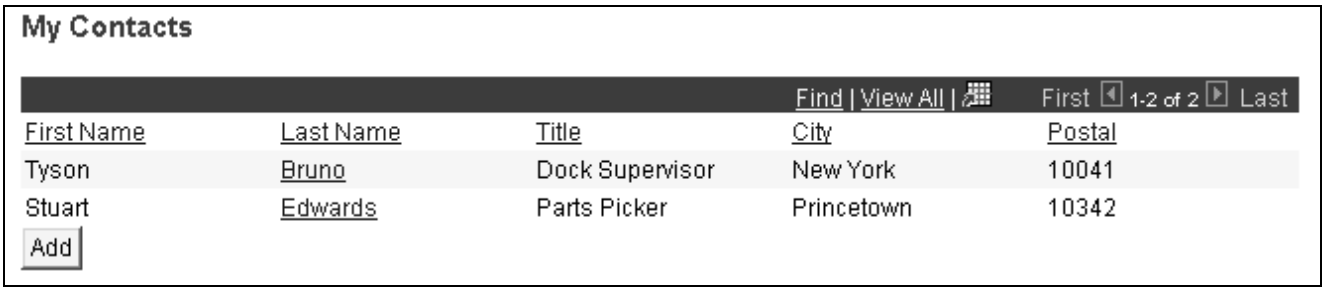
Description

This method returns the List View attributes for this property. These attributes describe display attributes for a column on a List View.

In the following screen shot, the columns include:

- First Name

- Last Name
- Title
- City
- Postal code



List View columns

Using the GetListViewAttrs method, you can return a reference to one of these columns, then using the ListViewAttrs properties, you can change how the column displays.

Parameters

Parameter	Description
propertyname	This is the name of the property for which the List View attributes are retrieved.

Returns

A ListViewAttrs object containing information about the property column.

Example

```
/* Change the Label for the City Property which is a member of the objects
in the QE_DEMO_SITE Collection  to "Town"*/
&ListAttrs = %ThisMobileObject.QE_DEMO_SITE.GetListViewAttrs("City");
&ListAttrs.Label = "Town";
```

Item

Syntax

Item(*index*)

Description

The **Index** method returns a reference to the mobile object referenced at *index*. This mobile object on the Mobile Page is the equivalent of one row on a level in the synchronizable Component Interface upon which the Mobile Page is based.

Parameters

Parameter	Description
index	Specify the index number for the mobile object to return. Indexes start at 1.

Returns

A reference to the mobile object that was returned.

Example

```
Local ApiObject &ob, &Session;
Local ApiObject &PropertyInfo;

&ob = %ThisMobileObject;
/*********************************************
* Insert some children                       *
*********************************************/
&ChildCount  = &ob.MyChild1.Count;

&ob.MyChild1.InsertItem(&ChildCount);
&ChildObject  = &ob.MyChild1.Item(&ChildCount + 1);

/*********************************************
* Manipulate Collections                      *
*********************************************/
&ob.MyChild1.InsertItem(0);
&ChildObject  = &ob.MyChild1.Item(1);
&ChildCount = &ob.MyChild1.Count;
If (&ChildCount > 0) Then;
   For &index = 1 To &ChildCount
     &ChildObject   = &ob.MyChild1.Item(&index);
     &PropertyInfo = &ChildObject.GetPropertyInfoByName("version");
     &PropertyInfo.Label  = "Version: ";
     &ob.MyChild1.Item(&index).version = &index + 100;
     &ob.MyChild1.Item(&index).GetPropertyInfoByName("version").visible = True;
   End-For;
End-If;
```

DeleteItem

Syntax

DeleteItem(*location*)

Description

The **DeleteItem** method removes the specified item from the collection and deletes the mobile object. This mobile object on the Mobile Page is the equivalent of one row on a level in the synchronizable Component Interface upon which the Mobile Page is based.

Note. Once you delete an item, it is deleted immediately. You no longer have access to it.

Parameters

Parameter	Description
location	Specify the location within the collection of the mobile object to delete. This parameter takes a numeric value.

Returns

A Boolean value: True if the mobile object was successfully deleted, False otherwise.

Example

```
Local ApiObject &ob, &Session;
Local ApiObject &PropertyInfo;

&ob = %ThisMobileObject;
/***********************************************
* Insert some children                         *
***********************************************/
&ChildCount  = &ob.MyChild1.Count;

&ob.MyChild1.InsertItem(&ChildCount);
&ChildObject  = &ob.MyChild1.Item(&ChildCount + 1);

/***********************************************
* Manipulate Collections                       *
***********************************************/
&ob.MyChild1.DeleteItem(1);
&ChildObject  = &ob.MyChild1.Item(1);
&ChildCount = &ob.MyChild1.Count;
```

InsertItem

Syntax

InsertItem(*location*)

Description

The **InsertItem** method inserts a new mobile object item at the specified location, plus 1. This mobile object on the Mobile Page is the equivalent of one row on a level in the synchronizable Component Interface upon which the Mobile Page is based.

Parameters

Parameter	Description
location	Specify the location of the new item. The mobile object is inserted at one past the number specified with *location*. For example, InsertItem(0) places the item in the first location, which is 1. InsertItem(count) places the new item in the last location: count + 1.

Returns

A Boolean value: True if the mobile object was successfully inserted, False otherwise.

Example

```
Local ApiObject &ob;

&ob = %ThisMobileObject;


If &MobileObjectValueExists = False Then
   &qe_mb_empl.create();
   /*Populate Level 1 data.  */
   &NewCICounter_LV1 = 0;
   &ob_top_parent = &ob.gettopparent();
   While &NewCICounter_LV1 <> &ob_top_parent.QE_MB_DEPARTMNT.count;
      /*Insert Level 1 Row  */
       &qe_mb_empl.QE_MB_DEPARTMNT.InsertItem(&NewCICounter_LV1 + 1);
       &qe_mb_empl.QE_MB_DEPARTMNT.Item(&NewCICounter_LV1 + 1).QE_MB_DEPARTMENT =
       &ob_top_parent.QE_MB_DEPARTMNT.Item(&NewCICounter + 1).QE_MB_DEPARTMENT_LV1;
       &qe_mb_empl.QE_MB_DEPARTMNT.Item(&NewCICounter_LV1 + 1).QE_MB_DEPT_DESCR =
       &ob_top_parent.QE_MB_DEPARTMNT.Item(&NewCICounter_LV1 + 1).QE_MB_DEPT_DESCR;
       &NewCICounter_LV2 = 0;
    End-While;
End-If;
```

Data Collection Properties

Following is the property for DataCollection.

Count

Description

The number of mobile objects in the collection. This is the equivalent of the number of rows on a level in the synchronizable Component Interface upon which the Mobile Page is based.

This property is read-only.

Example

```
Local ApiObject &ob, &Session;
Local ApiObject &PropertyInfo;

&ob = %ThisMobileObject;
/*********************************************
* Insert some children                       *
*********************************************/
&ChildCount  = &ob.MyChild1.Count;

&ob.MyChild1.InsertItem(&ChildCount + 1);
&ChildObject  = &ob.MyChild1.Item(&ChildCount + 1);

&ob.MyChild1.InsertItem(1);
&ChildObject  = &ob.MyChild1.Item(1);
&ChildCount = &ob.MyChild1.Count;
```

ListViewAttrs Class

The ListViewAttrs class objects are returned from the GetListViewAttrs data collection method.

The ListViewAttrs class has methods and properties that provide information about the properties for the synchronizable Component Interface upon which the Mobile Page is based.

ListViewAttrs Method

Following is the ListViewAttrs method.

IsSet

Syntax

IsSet(*attribute*)

Description

The IsSet method indicates if the given attribute had been explicitly set from PeopleCode, or if it is in an un-initialized state.

Parameters

Parameter	Description
attribute	Specify one of the ListViewAttrs properties, which contains display information about a mobile property column in a List View.

Returns

This method returns a Boolean value: True if this attribute has been set, False otherwise.

Example

```
/* Write the results to the results string  */

&results_string = "The following is " |
&ThisMobileObject.QE_DEMO_SITE("City").IsSet("Visible") |
": City is visible. "
```

ListViewAttrs Properties

Following are the ListViewAttrs properties.

DetailViewLabel

Description

Use this property to change the label shown at the top of a Mobile Page Detail View. DetailViewLabel is only available to a ListViewAttrs object that has been fetched with an empty string.

This property is read-write.

Example

Note that in this example, GetListViewAttrs uses an empty string as input. This is how to call GetListViewAttrs for use with DetailViewLabel.

```
/* Pass empty string */
&ListAttrs = %ThisMobileObject.QE_DEMO_SITE.Item(1).QE_DEMO_SITE_H.GetListViewAttrs⇒
("");
&ListAttrs.DetailViewLabel = "My Site Details";
```

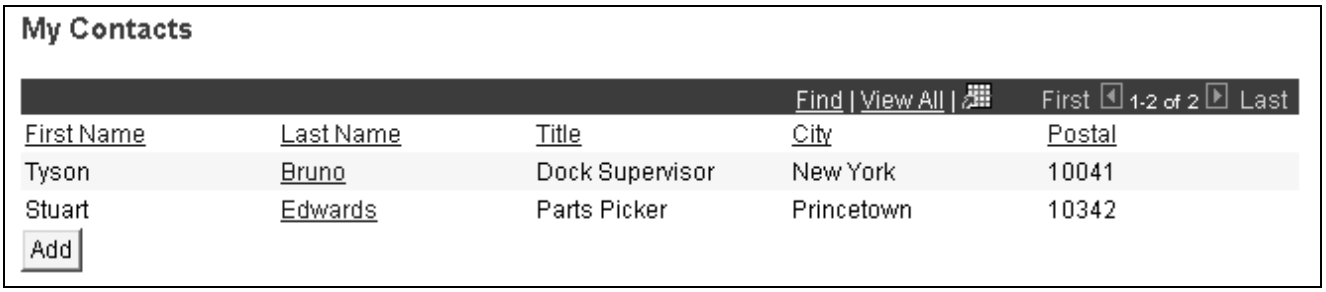
Label

Description

Use this property to change the lable shown at the top of a column in a List View; it is the lable for a property for mobile objects in a Data Collection.

This property is read-write.

In the following example, there are five columns, hence five labels: First Name, Last Name, Title, City and Postal.



List View columns

Example

```
/* Change the Label for the City Property which is a member of the objects
in the QE_DEMO_SITE Collection to "Town"*/
&ListAttrs = %ThisMobileObject.QE_DEMO_SITE.GetListViewAttrs("City");
&ListAttrs.Label = "Town";
```

ListViewLabel

Description

Use this property to change the tab label. The tab label is indirectly exposed through PeopleCode; it overwrites the List View Label, which is the default if no tab label is defined. However, if the ListViewLabel property is set for a property that has a tab label, the tab label is overridden with the PeopleCode ListViewLabel value.

ListViewLabel is only available to a ListViewAttrs object that has been fetched with an empty string.

This property is read-write.

Example

Note that in this example, GetListViewAttrs uses an empty string as input. This is how to call GetListViewAttrs for use with ListViewLabel.

```
/* Pass empty string */
&ListAttrs = %ThisMobileObject.QE_DEMO_SITE.Item(1).QE_DEMO_SITE_H.GetListViewAttrs⇒
("");
&ListAttrs.ListViewLabel = "My Sites";
```

Visible

Description

This property is True if this column for a property in a List View is visible. Setting this property to False hides the column.

Note. Even if a property has been marked as Invisible on the mobile page definition in PeopleSoft Application Designer, you can still override this property using PeopleCode.

This property is read-write.

Example

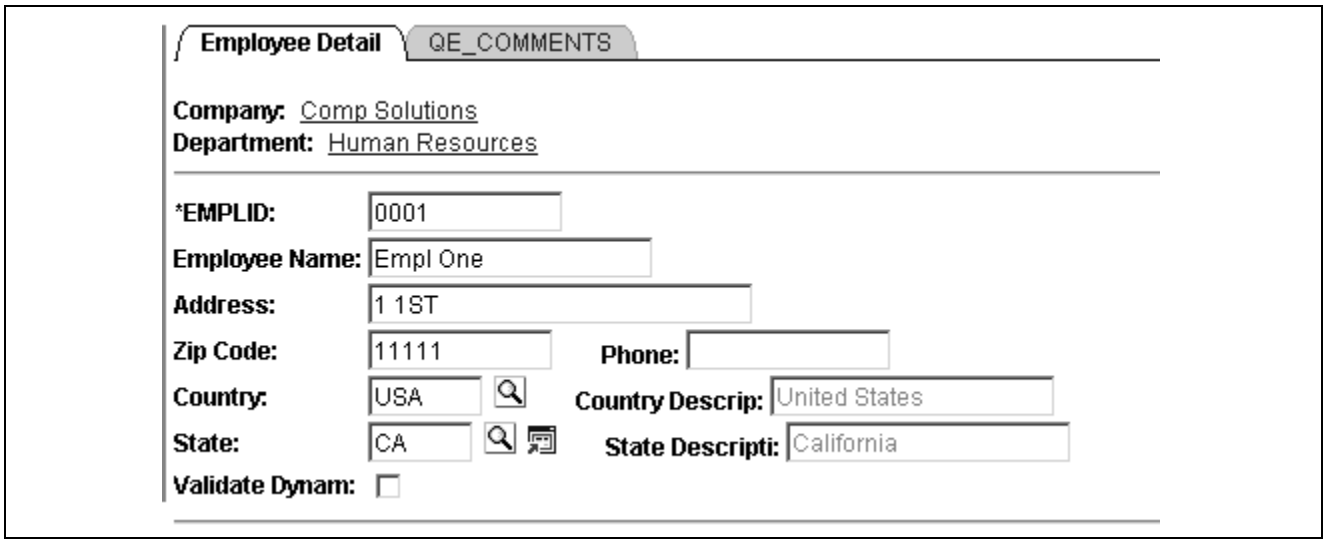
```
/* Make invisible the City property column which is a member of
the objects in the QE_DEMO_SITE Collection */

&ListAttrs = %ThisMobileObject.QE_DEMO_SITE.GetListViewAttrs("City");
&ListAttrs.Visible = False;
```

PropertyAttrs Class

The PropertyAttrs class objects are returned from the GetPropertyAttrsByName method. This class has methods and properties that provide information about the properties for the synchronizable Component Interface upon which the Mobile Page is based.

For example, to hide the QE_MB_ADDRESS property on the synchronizable Component Interface upon which the current Mobile Page and its mobile objects are based, you can use the following PeopleCode on an OnChange event.

```
%ThisMobileObject.GetPropertyAttrsByName("QE_MB_ADDRESS").Visible = False;
```



Mobile Page Detail View

PropertyAttrs Method

Following is the PropertyAttrs method.

IsSet

Syntax

IsSet(*attribute*)

Description

The **IsSet** method indicates if the given attribute had been explicitly set, or if it is in an un-initialized state.

Parameters

Parameter	Description
attribute	Specify one of the PropertyAttrs properties, which contains display information about a mobile property.

Returns

This method returns a Boolean: True if this attribute has been set, False otherwise.

Example

```
/*Write the results to the results field  */

&qe_mb_obj.QE_MB_RESULTS = "The following is " |
&qe_mb_obj_setup.GetPropertyAttrsByName(&CharacterPropertyName).IsSet("Visible") |
": " | GetPropertyAttrsByName(&CharacterPropertyName) | "is visible. ";
```

PropertyAttrs Properties

Following are the PropertyAttrs properties.

DisplayOnly

Description

This property is True if the mobile property is display only; False otherwise. This only affects the Mobile Page; this value is not saved to the mobile database and is not synchronized.

Note. If a property has been marked as Display Only on the mobile page definition in PeopleSoft Application Designer, you can override this property using PeopleCode. If a property has been marked as Display Only on the Component Interface, you can *not* override this property using PeopleCode.

If you use this property to make a collection display only, all the fiedls will be display only, except for those that are displayed as pushbuttons. To make a pushbutton display only, you must set the pushbutton property specifically to be display only, either in Application Designer or with PeopleCode. You must set the pushbutton property for every row to be display only.

This property is read-write.

Example

```
Declare Function Hide_or_Unhide_WorkingSetProps_SetValues PeopleCode QE_MB_PC_⇒
DER.QE_MB_GET FieldFormula;
```

```
/*If Get() checkbox is selected, unhide Set Value Properties, and grey
Create Check Box. Also maske sure Get() check box is ungrayed. */
If %ThisMobileObject.QE_MB_GET = True Then
   /*Make sure Create() Checkbox now has value "N"  */
   %ThisMobileObject.QE_MB_CREATE = False;
   Hide_or_Unhide_WorkingSetProps_SetValues( True);
   %ThisMobileObject.GetPropertyAttrsByName("QE_MB_CREATE").DisplayOnly = True;

Else
   /* If Get() checkbox is unselected, hide Set Value Properties, and ungrey
   Create Check Box. */
   If %ThisMobileObject.QE_MB_GET = False Then
      Hide_or_Unhide_WorkingSetProps_SetValues( False);
      %ThisMobileObject.GetPropertyAttrsByName("QE_MB_CREATE").DisplayOnly = False;
   End-If;
End-If;
```

InError

Description

Set this property within a SaveEdit event to cause any field on a Detail View that contains incorrect information to be displayed as red. This alerts the user to correct the data in that field.

This field is automatically reset. The developer does not have to reset this field.

This only affects the Mobile Page; this value is not saved to the mobile database and is not synchronized.

This property is read-write.

Example

```
%ThisMobileObject.GetPropertyAttrsByName("QE_MB_ADDRESS").InError = True;
```

Label

Description

Use this property to manipulate the label for this mobile property. This only affects the Mobile Page; this value is not saved to the mobile database and is not synchronized.

This property is read-write.

Example

```
&ob = %ThisMobileObject;
&PropertyInfo = &ob.GetPropertyAttrsByName("text");
&PropertyInfo.Label  = "This is text";
```

ShowRequiredCue

Description

An asterisk (*) is displayed beside fields that are defined as Required in PeopleSoft Application Designer. You can use this property to specify whether this asterisk, also called the **required field cue**, is displayed for a particular field.

This property is True if the mobile property displays the RequiredCue; False otherwise. This only affects the Mobile Page; this is not saved to the mobile database and is not synchronized.

Use this to make a field be required that was not specified as being required when the synchronized Component Interface was created in the PeopleSoft Application Designer. (You cannot make a required field be not required.)

For example, many fields are made required through PeopleCode. This means they aren't defined as Required in PeopleSoft Application Designer, and the end user may be confused. For these fields, you can use this property.

Note. This property affects only fields where a required field cue is otherwise permissible. That is, regardless of the setting of the property, no cue is ever shown on a push button, a display-only field, and so on.

This property is read-write.

Example

```
&ob = %ThisMobileObject;
&PropertyInfo = &ob.GetPropertyAttrsByName("text");
&PropertyInfo.ShowRequiredCue  = True;
```

Visible

Description

This property is True if this field is visible in the Mobile Page Detail View displaying it. Setting this property to False hides the field.

This only affects the Mobile Page; this value is not saved to the mobile database and is not synchronized.

Note. Even if a property has been marked as Invisible on the mobile page definition in PeopleSoft Application Designer, you can still override this property using PeopleCode.

If you set this value to False for a collection property, the collection is no longer visible.

Note. You can only make collections invisible using PeopleCode. You cannot set a collection to invisible in Application Designer.

This property is read-write.

Example

```
If %ThisMobileObject.QE_MB_PROP_METH = "Visible" Or
   %ThisMobileObject.QE_MB_PROP_METH = "DisplayOnly" Or
   %ThisMobileObject.QE_MB_PROP_METH = "ShowRequiredCue" Or
   %ThisMobileObject.QE_MB_PROP_METH = "SetModified" Then
   %ThisMobileObject.getpropertyattrsbyname("QE_MB_TRUE_FALSE").Visible = True;
End-If;

If %ThisMobileObject.QE_MB_PROP_METH = "Item" Or
   %ThisMobileObject.QE_MB_PROP_METH = "DeleteItem" Or
   %ThisMobileObject.QE_MB_PROP_METH = "InsertItem" Then
   %ThisMobileObject.getpropertyattrsbyname("QE_MB_PC_ITEM").Visible = True;
End-If;
```

PeerDefaultAttributes Class

PeerDefaultAttributes are returned by the GetPeerDefaultAttributes method of a mobile object.

When a peer reference does not have a value associated with it, the display attributes can only be accessed using the PeerDefaultAttributes class. After a peer reference has values, it can access using the ListViewAttrs class.

PeerDefaultAttributes Class Method

The following is the PeerDefaultAttributes class method.

IsSet

Syntax

IsSet(*Attribute*)

Description

The **IsSet** method indicates if the given attribute had been explicitly set, or if it is in an un-initialized state.

Parameters

Parameter	Description
Attribute	Specify one of the PeerDefaultAttributes properties, that contains display information about a peer reference.

Returns

A Boolean value: True if the attribute has been explicitly set, False otherwise.

PeerDefaultAttributes Properties

This section discusses the PeerDefaultAttributes properties in alphabetical order.

DisplayOnly

Description

This property is True if the peer reference is display only; False otherwise. This only affects the Mobile Page; this value is not saved to the mobile database and is not synchronized.

Note. If a property has been marked as Display Only on the mobile page definition in PeopleSoft Application Designer, you can override this property using PeopleCode. If a property has been marked as Display Only on the Component Interface, you can *not* override this property using PeopleCode.

This property is read-write.

Label

Description

Use this property to manipulate the label for this peer reference. This only affects the Mobile Page; this value is not saved to the mobile database and is not synchronized.

This property is read-write.

Visible

Description

This property is True if this peer reference is visible in the Mobile Page Detail View displaying it. Setting this property to False hides the peer reference.

This only affects the Mobile Page; this value is not saved to the mobile database and is not synchronized.

This property is read-write.

Classes and Functions and System Variables Used in Mobile

Mobile uses a subset of the classes, functions, and system variables used in PIA. Here is a list of those used in mobile.

PeopleCode Classes Used in Mobile

The following is a list of the PeopleCode classes used in mobile.

- Application Package

- Array
- Component Interface
- Exception
- Request (part of iScript)

Considerations Using the Request Class

Only the following request class method (%Request) is supported for mobile:

GetParameter

Only the following request class properties (%Request) are supported for mobile:

- BrowserType
- BrowserVersion
- HTTPMethod
- Protocol

Note. Other request class methods and properties may not return an error when sued in mobile, but do return a blank value, an empty array, False, and so on, according to the datatype of the value returned.

See Chapter 15, Internet Script Classes (iScript) , Request Class Methods , page 942.

System Variables Used in Mobile

The following is a list of the system variables used in mobile, by category.

- APIs: %Session, %Super, %This, %CompIntfcName
- Data and Time: %Date, %DateTime, %Request, %Time
- Mobile specific: %DeviceType, %MobilePage, %ThisMobileObject

Considerations Using Default Values

For a record field, you can specify constants for a default, such as %Date. Mobile only supports the following for record field defaults. In addition, these are only supported for date, datetime, and time type fields (as appropriate):

- %ClientDate
- %Date
- %DateTime
- %Time

See Also

Chapter 3, System Variables , page 563

Built-In Functions Used in Mobile

The following is a list of the built-in functions used in mobile, by category.

- Arrays: CreateArray, CreateArrayAny, CreateArrayRept, Split
- Conversion: Char, Code, String, Value
- Date and Time: AddToDate, AddToDateTime, AddToTome, Date, Date3, DatePart, DateTime6, DateTimeValue, DateValue, Day, Days, Days360, Days365, Hour, Minute, Month, Second, Time, Time3, TimePart, TimeValue, Weekday, Year
- Internet: GetHTMLText
- Language Constructs: Break, Declare Function, Evaluate, Exit, For, Function, If, Local, Repeat, Return, While
- Math: Abs, Acos, Asin, Atan, Cos, Cot, Degrees, Exp, Fact, Int, Integer, Ln, Log10, Nod, Product, Radians, Rand, Round, Sign, Sin, Sqrt, Tan, Truncate
- Message Catalog and Message Display: MessageBox, MsgGet, MsgGetExplainText, MsgGetText
- Mobile: TransferMobilePage, GenerateMobileTree
- OLE: CreateObject, ObjectDoMethod, ObjectGetProperty, ObjectSetProperty
- String: Clean, Code, Exact, Find, Left, Len, Lower, LTrim, Replace, Rept, Right, Rtrim, String, Substitute, Substring, Upper
- Validation: Error
- Workflow: Warning

See Also

Chapter 1, PeopleCode Built-in Functions, page 1

CHAPTER 19

Notification Classes

This chapter provides an overview of the Notify classes and discusses the following topics:

- Scope of the notification classes.
- Data types of the notification classes.
- How to import the notification classes.
- How to create a notification object.
- Notification class constructors.
- Notification class reference.

Understanding Notification Classes

The notification classes are used for ad-hoc notification. These classes enable you to create and send a notification to someone. The notifications can either be a Peoplesoft Worklist item or an SMTP email message. The notification class also stores notifications in a table.

The notification classes can be called from a component, an internet script, or an Application Engine program.

All of the classes, and most of the properties and methods that make up the Notification Classes have a GUI representation in Peoplesoft Workflow. This document assumes that the reader is familiar with PeopleSoft Workflow.

See Also

PeopleSoft Workflow, Introducing PeopleSoft Workflow

Scope of the Notification Classes

The notification classes can be instantiated only from PeopleCode.

The notification classes can be called from a component, an internet script, or an Application Engine program.

Notification classes can be of Local, Global, or Component scope.

Data Types of the Notification Classes

Every notification class is its own data type, that is, Notifications are declared as data type Notification, Notification addresses are declared as type NotificationAddress, and so on.

The following are the data types of the Notification Classes:

- Notification
- NotificationAddress
- NotificationTemplate
- WorklistEntry

How to Import the Notification Classes

The Notification Classes are *not* built-in classes, like Rowset, Field, Record, and so on. They are Application Classes. Before you can use these classes in your PeopleCode program, you must import them to your program.

An import statement names either all the classes in a package or one particular application class. For importing the Notification classes, PeopleSoft recommends that you import all the classes in the application package.

The application package PT_WF_NOTIFICATION contains the following classes:

- Notification
- NotificationAddress
- NotificationTemplate

The application package PT_WF_WORKLIST contains the WorklistEntry class.

The import statements you should use are as follows:

```
import PT_WF_NOTIFICATION:*;
import PT_WF_WORKLIST:*;
```

Using the asterisks after the package name makes all the application classes directly contained in the named package available. Application classes contained in subpackages of the named package are *not* made available.

See Also

Chapter 6, Application Classes , page 601

How to Create a Notification Object

After you've imported the Notification classes, you instantiate an object of one of those classes using the constructor for the class and the *Create* function.

The following example creates a new instance of the WorklistEntry class, as the variable &MyWE, with local scope:

```
Local WorklistEntry &MyWE = Create WorklistEntry();
```

See Also

Chapter 19, Notification Classes , Notification Classes Constructors , page 1063

Notification Classes Constructors

You must use the constructor for each class to instantiate an instance of that class. The following are the constructors for the Notification Classes.

Notification

Syntax

Notification(*NotifyFrom*, *dttmCreated*, *language_cd*)

Description

Use **Notification** to instantiate a notification object.

Parameters

Parameter	Description
NotifyFrom	Specify the entity the notification is being sent from, as a string. Avoid overusing generic entities with this parameter, as this value, along with the value for *dttmCreated* are the key to understanding where notifications originated from.
dttmCreated	Specify the datetime the notification is being sent, as a datetime value.
language_cd	Specify the 3-character language code of the current user, as a string.

Returns

A notification object.

Example

```
import PT_WF_NOTIFICATION:*;

import PT_WF_WORKLIST:*;

Local Notification &MyNotify;

&MyNotify = Create Notification("Workflow_Admin", %Datetime, %Language);
```

See Also

Chapter 19, Notification Classes , Notification Class, page 1066.

NotificationAddress

Syntax

NotificationAddress(*Oprid*, *Description*, *Language*, *EmailId*, *Channel*)

Description

Use **NotificationAddress** to instantiate a notification address.

This address is used to build a notification using the Notification class. This class is required if you want to use the Notification data type.

Parameters

Parameter	Description
Oprid	Specify the UserID of the person receiving the notification, as a string.
Description	Specify the description of the entity receiving the notification (distribution group names, people's names, and so on) as a string.
Language	Specify the language code of the entity to be notified as a string.
EmailId	Specify the email address as a string.
Channel	Specify the 'channel' used for the notification, as a string. The values are: • **Email**: notification is sent as email • **Worklist**: notification is sent as a worklist

Returns

None.

Example

```
import PT_WF_NOTIFICATION:*;

import PT_WF_WORKLIST:*;

Local NotificationAddress &MyNotify_Addy;

&MyNotify_Addy = Create NotificationAddress(%UserID, &Description, %Language,⇒
 &EmailID, &Channel);
```

See Also

Chapter 19, Notification Classes , NotificationAddress Class, page 1071.

NotificationTemplate

Syntax

NotificationTemplate(*ComponentId*, *Market*, *TemplateId*, *TemplateType*)

Description

Use **NotificationTemplate** to instantiate a notification template.

If you're specifying a generic template, the first two parameters do not have to have values, that is, you must specify a Null string ("") instead.

Parameters

Parameter	Description
ComponentId	Specify the name of the component the notification is coming from as a string. To specify a generic template, you must specify a Null string ("") for this value instead.
Market	Specify the name of the market of the component the notification is coming from as a string. To specify a generic template, you must specify a Null string ("") for this value instead.
TemplateId	Specify the template ID as a string.
TemplateType	Specify the type of template as a string. Values are: • G: generic template • C: component template

Returns

An instance of the NotificationTemplate class.

Example

```
import PT_WF_NOTIFICATION:*;

import PT_WF_WORKLIST:*;

Local NotificationTemplate &Notify_Temp;

/* create a generic template */

&Notify_Temp = Create NotificationTemplate("", "", "MyTemplate", "G");
```

See Also

Chapter 19, Notification Classes , NotificationTemplate Class, page 1072.

WorklistEntry

Syntax

WorklistEntry()

Description

Use **WorklistEntry** to instantiate a WorklistEntry object.

Parameters

None.

Returns

An instance of the WorklistEntry class.

Example

```
import PT_WF_NOTIFICATION:*;

import PT_WF_WORKLIST:*;

Local WorklistEntry &MyWorklistEntry = Create WorklistEntry();
```

See Also

Chapter 19, Notification Classes , WorklistEntry Class, page 1077.

Notification Class

This class is used to send a notification to people or people entities, as channel independent as possible. This enables functionality such as letting a user determine how to receive workflow notifications as a preference. It does not necessarily limit a person or entity to one channel.

See Chapter 19, Notification Classes , Notification Classes Examples , page 1084.

See *PeopleSoft Workflow*, Using Notification Templates.

Notification Class Import Statements

The Notification class imports the following classes:

- NotificationAddress
- WorklistEntry

Here are the import statements:

```
import PT_WF_NOTIFICATION:NotificationAddress;

import PT_WF_WORKLIST:WorklistEntry;
```

Notification Class Method

In this section, we discuss the Notification class method Send.

Send

Syntax

Send( )

Description

Use the **Send** method to send a notification to the entities defined in the NotifyTo, NotifyCC, and NotifyBCC, using the Subject and Message properties.

Parameters

None.

Returns

None.

See Also

PeopleSoft Workflow, Introducing PeopleSoft Workflow.

Notification Class Properties

In this section, we discuss the Notification class properties. The properties are discussed in alphabetical order.

ContentType

Description

This property returns the content type of the value specified by the Message property, as a string. This property is used only with email Notifications, that is, with SMTP email.

This property is read-write.

See Also

Chapter 19, Notification Classes , Message, page 1068.

dttmCreated

Description

This property returns the Datetime the notification created as a datetime value. Together with NotifyFrom this makes a notification unique.

See Also

Chapter 19, Notification Classes , NotifyFrom, page 1069.

EmailReplyTo

Description

If an email is sent as a result of the Send method, this property returns the email reply that should be used, as a string. This property can have a different value from NotifyFrom. This property is valid only with notifications that have "email" specified as the channel.

This property is read-write.

See Also

Chapter 19, Notification Classes , Send, page 1067, Chapter 19, Notification Classes , NotifyFrom, page 1069.

FileNames

Description

This property returns an array of string, populated with the names of any file attachments sent with the notification. This property is valid only with notifications that have "email" specified as the channel.

This property is read-write.

FileTitles

Description

This property returns an array of string, populated with descriptions of any file attachments. The array order *must* be respective to the array order for FileNames, that is, file description matches file name one by using the same array index value. This property is valid only with notifications that have "email" specified as the channel.

This property is read-write.

See Also

Chapter 19, Notification Classes , FileNames, page 1068.

language_cd

Description

This property returns the 3-character language code used for the notification, as a string. This is used for tracking purposes.

This property is read-write.

Message

Description

This property returns the text of the message sent with the notification as a string.

This property is read-write.

NotifyBCC

Description

This property returns an array of NotificationAddress objects, populated with the email addresses or worklist users of the people the notification is sent to. This list is used for notification information only, that is, the people on this list are generally not responsible for taking action specified by the notification. In addition, when you use the Notification Send method, other recipients cannot see the email addresses and or users that make up this array.

If you want to send a regular CC list, that is, one where the members of the list can see each other, use **NotifyCC** instead.

See Also

Chapter 19, Notification Classes , NotifyCC, page 1069.

NotifyCC

Description

This property returns an array of NotificationAddress objects, populated with the email addresses of the people the notification is sent to. This list is used for notification information only, that is, the people on this list are generally not responsible for taking action specified by the notification. When using the Notification Send method, the email address or worklist users of this array are exposed to the recipients of the notification.

If you want to send a blind-CC list, that is, where the members of the list cannot see each other, use **NotifyBCC** instead.

See Also

Chapter 19, Notification Classes , NotifyBCC, page 1069.

NotifyFrom

Description

This property returns the ID of the entity sending the notification as a string. For example, ad-hoc workflow uses the current UserID.

This property is read-write.

NotifyGuid

Description

This property returns the GUID of the notification. This is a unique identifier used for tracking purposes only.

This property is read-write.

NotifyTo

Description

This property returns an array of NotificationAddress objects, populated with the email addresses of the people the notification is sent to, and who are responsible for taking action if necessary.

To send the list to additional people, use the NotifyCC or NotifyBCC properties.

See Also

Chapter 19, Notification Classes , NotifyCC, page 1069, Chapter 19, Notification Classes , NotifyBCC, page 1069.

SourceComponent

Description

This property returns the name of the source component, that is, that initiated the notification process. This property is used for transaction tracking purposes only. This property is not required.

This property is read-write.

SourceMarket

Description

This property returns the name of the market of the source component, that is, that initiated the notification process. This property is used for transaction tracking purposes only. This property is not required.

This property is read-write.

SourceMenu

Description

This property returns the name of the menu of the source component, that is, that initiated the notification process. This property is used for transaction tracking purposes only. This property is not required.

This property is read-write.

Subject

Description

This property returns the subject used in the email notification as a string.

This property is read-write.

Template

Description

If the notification message originated from a template, this property returns the name of that template, as a string. This property is used for transaction tracking purposes only. This property is not required.

This property is read-write.

NotificationAddress Class

The main purpose of the NotificationAddress is to define an entity to a ’channel’, it is concerned at a high level only that a notification was sent.

See Chapter 19, Notification Classes , Notification Classes Examples , page 1084.

See *PeopleSoft Workflow*, Using Notification Templates.

NotificationAddress Class Properties

In this section, we discuss the NotificationAddress class properties. The properties are discussed in alphabetical order.

Channel

Description

This property returns the ’channel’ used for the notification, as a string. The values are:

Value	Description
Email	Notification is sent as email.
Worklist	Notification is sent as a worklist.

This property is read-write.

Description

Description

This property returns the description of the entity as a string. Entity descriptions include distribution group names or people’s names.

This property is read-write.

EmailId

Description

This property returns the Email Address of the entity to be notified.

This property is read-write.

Language

Description

This property returns the language code of the entity to be notified.

This property is read-write.

Oprid

Description

This property returns the UserID of the person receiving the notification as a string.

This property is read-write.

NotificationTemplate Class

This class is useful for populating the message property of the Notification class.

See *PeopleSoft Workflow*, Using Notification Templates.

NotificationTemplate Class Methods

In this section, we discuss the NotificationTemplate class methods. The methods are discussed in alphabetical order.

GetAndExpandTemplate

Syntax

GetAndExpandTemplate(*Language, Vars*)

Description

Use the **GetAndExpandTemplate** method to expand and map the variables into the Text property of the template.

If this is a component template, you must use the **SetupCompVarsAndRcpts** method first, before you use this method.

If this is a generic template, you must use the **SetupGenericVars** method first, before you use this method.

The value returned by the Setup methods must be used as the second parameter for this method.

Parameters

Parameter	Description
Language	Specify the 3-character language code used with the notification.
Vars	Specify the variable representing the XML string returned by either the SetupCompVarsAndRcpts method or the SetupGenericVars method.

Returns

Returns a Boolean value: True if the template expanded properly, False otherwise.

Example

```
import PT_WF_NOTIFICATION:NotificationTemplate;

Local NotificationTemplate & mynotifytemplate;

&mynotifytemplate = create NotificationTemplate(%Component, %Market, ⇒
&templateName, "C");

&xmlVars = &mynotifytemplate.SetupCompVarsAndRcpts(GetLevel0());

&mynotifytemplate.GetAndExpandTemplate(%Language, &xmlVars);
```

See Also

Chapter 19, Notification Classes , SetupCompVarsAndRcpts, page 1073, Chapter 19, Notification Classes , SetupGenericVars, page 1074.

SetupCompVarsAndRcpts

Syntax

SetupCompVarsAndRcpts(*&Rowset_Context*)

Description

Use the **SetupCompVarsAndRcpts** method to set up variables for the component from the provided rowset. This method also sets up any additional recipients defined in the component.

The rowset corresponds to the component that the notification template was instantiated with. For example, if you created a notification template for JOB, the rowset used with this method should be based on the JOB component.

If you want to expand a component template, you must use this method first, before expanding the template.

Parameters

Parameter	Description
&Rowset_Context	Specify an already instantiated and populated rowset containing the necessary variables.

Returns

A string containing XML that is used by the GetAndExpandTemplate method.

Example

```
import PT_WF_NOTIFICATION:NotificationTemplate;

Local NotificationTemplate & mynotifytemplate;

&mynotifytemplate = create NotificationTemplate(%Component, %Market, ⇒
&templateName, "C");

&xmlVars = &mynotifytemplate.SetupCompVarsAndRcpts(GetLevel0());

&mynotifytemplate.GetAndExpandTemplate(%Language, &xmlVars);
```

See Also

Chapter 19, Notification Classes , GetAndExpandTemplate, page 1072.

SetupGenericVars

Syntax

SetupGenericVars(*&AryValues*)

Description

Use the **SetupGenericVars** method to pass values to a generic template.

To expand a generic template, you must use this method first, before expanding the template.

Parameters

Parameter	Description
&AryValues	Specify an array of string containing values used with the expanded template.

Returns

A string containing XML that is used by the GetAndExpandTemplate method.

Example

```
import PT_WF_NOTIFICATION:NotificationTemplate;

. . .
```

```
&mynotifytemplate = create NotificationTemplate("","", "MYTEMPLATE", "G");

/* Populate an array to contain the values needed by */
/* the template */
&aryValues = CreateArrayRept("", 0);
&aryValues.Push("FIRST VALUE");
&aryValues.Push("SECOND VALUE");
&xmlVars = &mynotifytemplate.SetupGenericVars(&aryValues);
&mynotifytemplate.GetAndExpandTemplate(%Language, &xmlVars);
```

See Also

Chapter 19, Notification Classes , GetAndExpandTemplate, page 1072.

NotificationTemplate Class Properties

In this section, we discuss the NotificationTemplate class properties. The properties are discussed in alphabetical order.

ComponentId

Description

This property returns a component for the component type notification template as a string. This is not required for generic templates.

This property is read-write.

Instruction

Description

This property returns the notification instructions as a string.

This property is read-write.

Language

Description

This property returns the language used to retrieve template descriptions, as a string.

This property is read-write.

Market

Description

This property returns the market for component type notification templates as a string. This is not required for generic templates.

This property is read-write.

Priority

Description

This property returns the priority of the notification template as a string.

This property is read-write.

Responses

Description

This property is populated only if there are any RIMM responses.

This property is read-write.

Subject

Description

This property returns the subject of the notification template as a string.

This property is read-write.

TemplateId

Description

This property returns the template ID as a string. If you are creating a notification template, this property is always required.

This property is read-write.

TemplateType

Description

This property returns the template type as a string. Values are:

Value	Description
C	Component
G	Generic

If you are creating a notification template, this property is always required.

This property is read-write.

Text

Description

This property returns the Template Text as a string.

This property is read-write.

WorklistEntry Class

This class is used to create and or update a worklist entry.

See Chapter 19, Notification Classes , Creating a WorklistEntry , page 1084.

See Chapter 19, Notification Classes , Updating a WorklistEntry , page 1086.

See *PeopleSoft Workflow*, Introducing PeopleSoft Workflow.

WorklistEntry Methods

In this section, we discuss the WorklistEntry class methods. The methods are discussed in alphabetical order.

Create

Syntax

```
Create()
```

Description

Use the **Create** method to create a new WorklistEntry object.

This method is *not* the same as the Create *constructor*. The Create constructor only creates an instance of the WorklistEntry class. To add the entry to the database, you must use this method.

The following properties must be set before calling this method:

- busactivity
- buseventname
- busprocname
- worklist

If this method completes successfully, the following properties are populated:

- instanceid
- transactionid

Parameters

None.

Returns

This method returns a numeric value: 0 if there is an error, nonzero if entry created successfully.

Example

```
import PT_WF_WORKLIST:*;

Local WorklistEntry &worklist;
Local Number &Rslt;

&worklist = create WorklistEntry();

&worklist.busprocname  = "Administer Workflow";
&worklist.busactivity  = "Send Note";
&worklist.buseventname = "Worklist Note";
&worklist.worklistname = "Worklist Note";

&Rslt = &worklist.Create();
```

See Also

Chapter 19, Notification Classes , busactivity, page 1080, Chapter 19, Notification Classes , buseventname, page 1080, Chapter 19, Notification Classes , busprocname, page 1080, Chapter 19, Notification Classes , instanceid, page 1081, Chapter 19, Notification Classes , transactionid, page 1083.

Save

Syntax

Save()

Description

Use the **Save** method to save a WorklistEntry to the database.

PeopleCode Event Considerations

You must include this method within events that allow database updates. This includes the following PeopleCode events:

- SavePreChange (Page)
- SavePostChange (Page)
- Workflow
- Message Subscription

- FieldChange

If this method results in a failure, all database updates are rolled back. All information the user entered into the component is lost, as if the user pressed ESC.

Parameters

None.

Returns

A numeric value: 0 if save didn't complete successfully, nonzero if save completed successfully.

Example

```
If (&worklist.Save() <> 0) Then
    /* success */
Else
   /* handle error */
End-If;
```

See Also

Chapter 19, Notification Classes , Update, page 1079.

Update

Syntax

Update()

Description

Use the **Update** method to update the worklist entry with any property values that have changed, and assign values to the associated record in the database. However, this method does *not* commit the changes to the record in the database. You must use the **Save** method to update the database.

Parameters

None.

Returns

A numeric value: 0 if update didn't complete successfully, nonzero if update completed successfully.

Example

```
If (&worklist.Update() <> 0) Then
  &worklist.inststatus = 2  /* mark entry worked */
  /* additional processing */
End-if;
```

See Also

Chapter 19, Notification Classes , Save, page 1078.

WorklistEntry Properties

In this section, we discuss the WorklistEntry class properties. The properties are discussed in alphabetical order.

actiondttm

Description

This property returns the Worklist action datetime as a datetime.

This property is read-write.

busactivity

Description

This property returns the source activity name of the source business process as a string.

This property is read-write.

buseventname

Description

This property returns the source event name of the source activity as a string.

This property is read-write.

busprocname

Description

This property returns the source business process name as a string.

This property is read-write.

commentshort

Description

This property returns the short comment the previous UserID gave when reassigning a worklist as a string.

This property is read-write.

do_replicate_flag

Description

This property returns a flag that indicates if the worklist entry must be replicated. Values are:

Value	Description
Y	Worklist entry needs to be replicated
N	Worklist entry does not need to be replicated

This property is read-write.

instanceid

Description

This property returns the Instance Id. This property is used with the following properties to make the worklist entry a unique number:

- busprocname
- busactivity
- buseventname
- worklistname

This property is read-write.

instselecteddttm

Description

This property returns the datetime that the worklist was selected to be worked as a datetime value.

This property is read-write.

instatus

Description

This property returns the Worklist Entry status as a string. The values are:

Value	Description
0	new worklist entry
1	selected worklist entry
2	worked worklist entry
3	cancel string

This property is read-write.

insttimeoutddttm

Description

This property returns the datetime the worklist entry timed out. Only some worklist types use this property.

This property is read-write.

instworkeddttm

Description

This property returns the datetime worklist was worked as a datetime value.

This property is read-write.

oprid

Description

This property returns the UserID that is assigned the worklist as a string.

This property is read-write.

originatorid

Description

This property returns the UserID that caused the worklist to be created as a string.

This property is read-write.

prevoprid

Description

This property returns the previous UserID who used this worklist entry as a string.

This property is read-write.

syncid

Description

This property is reserved for future use.

This property is read-write.

timedout

Description

This property returns the status flag if the worklist has timed out. Values are:

Value	Description
Y	Worklist has timed out
N	Worklist has not timed out

This property is read-write.

transactionid

Description

This property returns the transaction Id as a number, which represents the unit of work. This can be useful for pooled worklists.

This property is read-write.

url

Description

This property returns the URL to work for the worklist as a string. This is used only for replicating worklist entries across PeopleSoft databases.

This property is read-write.

wl_priority

Description

This property returns the priority of the worklist as a string. The values are:

Value	Description
low	Worklist has low priority.
medium	Worklist has medium priority.
high	Worklist has high priority.

This property is read-write.

wldaystoselect

Description

This property returns the number of days to select worklist as a number.

This property is read-write.

wldaystowork

Description

This property returns the number of days to work the worklist as a number.

This property is read-write.

worklistname

Description

This property returns the source worklist name for the source event string as a string.

This property is read-write.

Notification Classes Examples

The following are example of typical actions you perform using the notification classes.

See *PeopleSoft Workflow*, Using Notification Templates.

Creating a WorklistEntry

The following example creates a WorklistEntry.

The following is the complete code sample: the steps explain each line.

```
import PT_WF_WORKLIST:*;

Local WorklistEntry &worklist;

&worklist = create WorklistEntry();

&worklist.busprocname  = "Administer Workflow";
&worklist.busactivity  = "Send Note";
&worklist.buseventname = "Worklist Note";
&worklist.worklistname = "Worklist Note";

If (&worklist.Create() <> 0) Then
  &worklist.oprid = %UserId;
  &worklist.originatorid = %UserId;
Else
  /* handle error */
End-If;

If (&worklist.Save() <> 0) Then
  /* success, create a corresponding row in */
  /* application worklist record... */
Else
```

```
  /* handle error */
End-If;
```

To create a WorklistEntry:

1. Import the WorklistEntry class.

   You must import the WorklistEntry class before you can use it in your PeopleCode program. In addition, the second line declares the variable &worklist as type WorklistEntry.

   ```
   import PT_WF_WORKLIST:*;

   Local WorklistEntry &worklist;
   ```

2. Instantiate a WorklistEntry object.

   Using the constructor, create an instance of the WorklistEntry class. This does *not* create a WorklistEntry. This creates a WorklistEntry object only. You must use the Create method to actually create the WorklistEntry in the system.

   ```
   &worklist = create WorklistEntry();
   ```

3. Set the required parameters.

   Some of the WorklistEntry class properties are required for every WorklistEntry. You must set a value for these properties before you can update or save the WorklistEntry.

   ```
   &worklist.busprocname  = "Administer Workflow";
   &worklist.busactivity  = "Send Note";
   &worklist.buseventname = "Worklist Note";
   &worklist.worklistname = "Worklist Note";
   ```

4. Create the WorklistEntry.

   After you've set the required values, you need to create the WorklistEntry in the system. This does *not* update the database. This does, however, add values to the notification tables.

   If this method completes successfully, the following properties are populated: instanceid and transactionid.

   - instanceid
   - transactionid

   ```
   If (&worklist.Create() <> 0) Then
     &worklist.oprid = %Userid;
     &worklist.originatorid = %Userid;
   Else
     /* handle error */
   End-If;
   ```

5. Save the WorklistEntry.

   After you've created the WorklistEntry, save it. The Save method actually updates the database, and so can be used only in events that allow database updates.

   ```
   If (&worklist.Save() <> 0) Then
     /* success, create a corresponding row in */
     /* application worklist record... */
   ```

```
Else
  /* handle error */
End-If;
```

Updating a WorklistEntry

The following example updates a WorklistEntry so that it is considered worked.

The following is the complete code sample: the steps explain each line.

```
import PT_WF_WORKLIST:*;

Local WorklistEntry &worklist;

&worklist = create WorklistEntry();

&worklist.busprocname  = "Administer Workflow";
&worklist.busactivity  = "Send Note";
&worklist.buseventname = "Worklist Note";
&worklist.worklistname = "Worklist Note";
&worklist.instanceid = 101;

If (&worklist.Update() <> 0) Then
  &worklist.inststatus = 2  /* mark entry worked */
  If (&worklist.Save() <> 0) Then
   /* success */
  Else
   /* handle error */
  End-If;
Else
  /* handle error */
End-if;
```

To update a WorklistEntry:

1. Import the WorklistEntry class.

   You must import the WorklistEntry class before you can use it in your PeopleCode program. In addition, the second line declares the variable &worklist as type WorklistEntry.

   ```
   import PT_WF_WORKLIST:*;

   Local WorklistEntry &worklist;
   ```

2. Instantiate a WorklistEntry object.

   Using the constructor, create an instance of the WorklistEntry class. This does *not* create a WorklistEntry. This creates a WorklistEntry object only. You must use the Create method to actually create the WorklistEntry in the database tables, and the Save method to update the tables.

   ```
   &worklist = create WorklistEntry();
   ```

3. Set the required parameters.

Some of the WorklistEntry class properties are required for every WorklistEntry. You must set a value for these properties before you can update or save the WorklistEntry. The instanceid property, combined with the other properties, indicates a unique WorklistEntry. This value is populated after a WorklistEntry has already been created.

```
&worklist.busprocname  = "Administer Workflow";
&worklist.busactivity  = "Send Note";
&worklist.buseventname = "Worklist Note";
&worklist.worklistname = "Worklist Note";
&worklist.instanceid = 101;
```

4. Update and save the WorklistEntry.

   After you've set the required parameters, try to update the WorklistEntry. If that is successful, mark the WorklistEntry as 'worked', and try to save the WorklistEntry. The Save method actually updates the database, and so can be used only in events that allow database updates.

```
If (&worklist.Update() <> 0) Then
  &worklist.inststatus = 2  /* mark entry worked */
  If (&worklist.Save() <> 0) Then
   /* success */
  Else
   /* handle error */
End-If;
Else
   /* handle error */
End-if;
```

CHAPTER 20

Page Class

This chapter provides an overview of Page class and discusses the following topics:

- Shortcut considerations.
- Page object declaration.
- Scope of a Page object.
- Page class built-in function.
- Page class reference.

Understanding Page Class

Use the page class to manipulate a page in a component. The most common use of this class is to hide or display a page in a component, either based on field values or the level of security of the user.

Generally, the PeopleCode used to manipulate a page object is associated with PeopleCode in the Activate event.

Note. The page object should not be used until after the page processor has loaded the page: do not instantiate this object in RowInit PeopleCode, use it in PreBuild, PostBuild or Activate instead.

In addition, if you need to hide the current page, PeopleSoft recommends using the PreBuild event for your program.

See Also

PeopleCode Developer's Guide, PeopleCode and the Component Processor, Component Build Processing in Update Modes

Shortcut Considerations

An expression of the form

```
PAGE.pagename.property
```

is equivalent to **GetPage**(**PAGE**.*name*).*property*. For example, the following two lines of code are equivalent:

```
PAGE.MANAGER_APPROV.DisplayOnly = False;
```

```
GetPage(PAGE.MANAGER_APPROV).DisplayOnly = False;
```

Data Type of a Page Object

Use the Page data type to declare Page objects. For example,

```
Local Page &MYPAGE;
```

Scope of a Page Object

A page object can be instantiated from PeopleCode only.

Use this object only in PeopleCode programs that are associated with an online process, not in an Application Engine program, a message subscription, a Component Interface, and so on.

In addition, the page object should not be used until after the page processor has loaded the page: do not instantiate this object in RowInit PeopleCode. Use it in PostBuild or Activate instead.

Page Class Built-in Function

Chapter 1, PeopleCode Built-in Functions, GetPage , page 262

Page Class Properties

In this section, we discuss each Page class properties.

DisplayOnly

Description

Use this property to make a page display-only, or to enable a page so the fields can be edited. This property takes a Boolean value: True if the page is display-only, False otherwise.

Note. You cannot enable or disable the current page.

The value of DisplayOnly for a page is restored to the value set in PeopleSoft Application Designer when the user goes to another component. This means you must set DisplayOnly for the page each time a component is accessed.

This property is read-write.

Example

```
If &MYPAGE.DisplayOnly Then
  &MYPAGE.Visible = True;
Else
  &MYPAGE.Visible = False;
End-If;
```

Name

Description

This property returns a reference to the page name definition (a string value.)

This property is read-only.

Example

```
&MYPAGENAME = &MYPAGE.Name;
```

Visible

Description

If this property is True, the page executing this property is displayed. If you set this property to False, the page is hidden. When the visibility is changed, the component tabs and menus are also automatically changed to reflect the new settings. Menus return to their original values when the user goes to another component.

If you set a page to be hidden in the component definition, you can change the value of the page to be visible at runtime.

PeopleSoft recommends using the PreBuild event if you need to hide the current page.

Example

In the following example, a page is hidden based on the value of the current field.

```
If PAYROLE_TYPE = "Global" Then
  PAGE.JOB_EARNINGS.Visible = False;
End-If;
```

See Also

PeopleSoft Application Designer, Creating Component Definitions.

CHAPTER 21

PortalRegistry Classes

This chapter provides an overview of PortalRegistry classes and discusses how to:

- Use the PortalRegistry API
- Use security
- Work with ValidFrom and ValidTo
- Do error handling
- Understand the life-cycle of a PortalRegistry object
- View the PortalRegistry class hierarchy
- Use content references
- Understand naming conventions
- Delete content considerations
- Save content considerations
- Declare a PortalRegistry object
- Understand the scope of a PortalRegistry object
- Use PortalRegistry (reference section)
- Understand PortalRegistry classes examples

See Also

PeopleTools Internet Technology, Understanding PeopleSoft Portal Technology

Understanding Portal Registry

The portal registry is a tree structure in which content for a portal is organized, classified, and registered. The PortalRegistry classes (API) are used to update the portal registry. The Portal Administration pages provide a GUI access to the PortalRegistry API. You can also access them using a PeopleCode program you write yourself. How you access the portal registry depends on the type of updates required. Your organization will likely use both methods of updating the portal registry. This chapter focuses on accessing the PortalRegistry classes using PeopleCode.

A portal is a website that helps you navigate to other web-based applications and content. The PeopleSoft Portal is a *business portal*. It is similar to general purpose portals, except that its main purpose is to help end-users be more effective in accessing information to perform their jobs.

As a licensee of PeopleTools, you are licensed to use the base portal technology, which is limited to navigation to licensed PeopleSoft applications. If you want to register additional non-PeopleSoft content, customize your homepage, or create any pagelets, you must license PeopleSoft Enterprise Portal.

Each PeopleSoft Portal is defined by one PeopleSoft portal registry. The PeopleSoft portal registry consists of a number of system tables and associated data in a PeopleSoft database. The portal registry must reside within one PeopleSoft database. There can be more than one portal registry in a PeopleSoft database, but only one portal registry is associated with a PeopleSoft Portal.

The portal registry consists of the following primary parts:

- Folders
- Content references
- Nodes

Folders and content references make up the majority of the portal registry, and provide a hierarchical tree structure to describe various content that is *registered* as part of a PeopleSoft portal.

Nodes provide a logical name to a specific webserver and database, so content can be registered independent of specific webservers. It is used when the portal servlet attempts to retrieve content—both internal PeopleSoft content as well as external content—and to assemble pages.

The primary function of the portal is to take a target URL (generally a URL for a PeopleSoft component) that comes in from a user's browser, and assemble a page with that content and any other content. The layout and what content to assemble on the page is defined by the *node template*, which is composed of HTML. The portal attempts to find the content reference associated (that is, registered) with the target URL to get the template, or uses a series of default templates if it cannot get the template from a content reference.

Folders

Folders are how a hierarchical structure is created within a portal registry. Each folder has a parent, except the root folder, which is the top-level folder in a portal registry. Each folder can also contain child folders and content references. Folders are roughly analogous to directories within a file system. A folder can be further defined by a number of attributes (description, security, when it expires, and so on) that are useful within the portal environment.

See Also

Chapter 21, PortalRegistry Classes , Adding a Folder, page 1248

Content References

A content reference is simply a reference to a URL. After you create an entry for a content reference in the portal registry, it's considered registered. A content reference can be further defined by a number of attributes (description, security, when it expires, and so on) that are useful within the portal environment.

There are a number of distinct types of content references that can be broken down into the following broad categories:

- Target
- Template

- Component

A *target* type of content reference describes a registered URL that a user might reference. Typically, these would be a PeopleSoft Internet Architecture component, such as a page in a transaction. When a user references a URL, the URL is looked up in the registry to find the target content reference.

A *template* type of content reference describes what, if any, other content to put on the page, and where to place that content. The portal attempts to find a template for every URL that is requested.

A *component* type of content reference describes a component that is typically placed on a target page or homepage.

Nodes

When a content reference is created to register some content (that is, a URL) the specific URI (that is, the scheme, webserver, and so on) can be specified in the content reference. However, this has the drawback that every time the URI for a content reference changes (the webserver name changes, and so on), the content reference must be changed. Nodes are a way to create a logical name for a specific webserver, scheme, and so on. When the content reference is created you can specify a node name.

For example, suppose the name of a webserver changed. If you don't use nodes, you must check all your content references and change them accordingly. If you use nodes, you have to change only the webserver name in one place, and all the content references that use that node now automatically reference the correct webserver.

Nodes can be optional for content where the specified URL is already a complete URL, such as for external content. Nodes can be categorized as:

- default local
- local
- non-local (remote)

See Also

Chapter 21, PortalRegistry Classes , Using Attributes, page 1255

Security

The same security mechanism is used for folders, content references, tab definitions, pagelets, and user homepages. All of these items can be marked as public, owner accessible, or can have explicit PeopleSoft permission values set, including cascading the permissions to its child objects (that is, the child objects have the same permissions as the parent objects, when applicable.)

When marked as public, the item is accessible by any user. Public access cannot be cascaded, that is, passed down, to child objects.

When marked as owner accessible, an item is accessible only by the same user that created that item. Owner accessible cannot be cascaded, that is, passed down, to child objects.

Items can be marked as accessible by PeopleSoft permissions. This means that if a user is a member of a role, and the role contains the permission list that the item is also associated with, the user has access to that item. Additionally, when applied to folder permissions the permission can be cascaded. This means that any child of that folder, including a content reference, automatically has that permission added to its permission list.

Attributes

Folders, content references, PageletCategory objects, and Pagelets can have any number of attributes added to them. Attributes are simply name/value pairs. These name/value pairs are defined and used by many portal-aware applications, such as the search engine, navigational display, and so on.

See Chapter 21, PortalRegistry Classes , Using Attributes, page 1255.

Additional Portal Objects

Using the PortalRegistry, you can also add and customize other items in your portal, such as tabs, homepages, and favorites.

The TabDefinition is a homepage tab that has been defined to the portal. It is what an administrator creates when they want to define a new homepage tab. The TabDefinition is based on content references.

The User Homepage is a personalized homepage for a user. It contains all the tabs and pagelets the user has selected for their homepage. The User Homepage is based on folders.

A Favorite represents a content reference that the user wants to keep a shortcut to. It contains the name of the content reference, and the label the user wants displayed for this favorite.

A Pagelets is an area on a page that contains content, and the template used to specify the style for that content. Pagelets are a type of content reference.

Using the PortalRegistry API

The PortalRegistry API provides the entry point to a specific portal registry. Common administrative tasks include adding, deleting, and renaming the registry objects (that is, folders, content references, tabs, and so on.) Additionally, there are many properties associated with every registry object, and all of these properties can be accessed and modified.

You can use the PortalRegistry API in batch mode to make many changes to a portal at once. For example, suppose something changed in your security system. You could write your own PeopleCode program, using the PortalRegistry classes, in an Application Engine program and change the access for all your users at once.

You can also use the PortalRegistry API to exchange data with an external system. For example, suppose an external merchant had an area on your company's portal, and the information there must be updated. One way to do this is for the merchant to use an Application Message to send the data to your system, then a subscription program would update the portal using the PortalRegistry API.

Each PortalRegistry object represents one specific PortalRegistry in a system. An empty PortalRegistry object is initially gotten from the PeopleSoft Session object. You can then open an existing PortalRegistry to view or change existing content, create a new PortalRegistry, or delete an existing PortalRegistry.

Using Security

The PermissionValue object associated with a folder, content reference, PageletCategory object or Pagelet, as well as specific properties on the object, work together to form the security for an object. In this section, we discuss how to:

- Use object properties.
- Access objects.

Using Object Properties

The properties that set permissions for an object are:

- AuthorAccess
- PublicAccess

The AuthorAccess property determines whether the author of the object has access to the object.

The PublicAccess property determines whether the object is generally accessible or not. If this property is set to True, all users have access to the object.

These properties apply only to an object. They *cannot* be cascaded, that is, passed down to child objects.

The other object property used with security, the Authorized property, indicates whether a user is authorized to access an object. The value of this property depends on whether the user has access.

Accessing Objects

When you get a folder, content reference, PageletCategory or Pagelet collection, only the items that the end-user is authorized to access are in the collection.

An object is contained in a collection is based on the following algorithm.

- If the object is marked as PublicAccess it is automatically accessible.
- If the object is marked as AuthorAccess and the current UserId is the Author it is accessible.
- If the current user is in a permission list (class) that is in the Permissions collection for that object.
- If the current user is in a permission list (class) that is in the CascadedPermissions collection for that object.
- If the current user has the Portal Administrator role.

When you access a content reference or folder using a valid name and one of the Find methods (FindCRefByURL, FindCRefByName, or FindFolderByName) a content reference or folder is returned whether the user is authorized to it. When you use these methods, always check the Authorized property. This is the only property you can view from an object that an end-user isn't authorized to.

In addition, if you know the specific URL for a tab, you could specify that in the FindCRefByURL. The tab is returned whether the user is authorized to it, so you should always check the Authorized property.

Using PermissionValue and Cascading Permissions

To set permissions for a folder, content reference, PageletCategory object, or Pagelet, you must access the PermissionValue collection for that object using the Permissions property. The PermissionValue objects refer to permission list values that already exist on the system, such as ALLPANLS, CUSTOMER, EMPLOYEE, and so on.

For every PermissionValue object, you can chose whether all the child folders and content references of this folder have the same permissions. This is called *cascading*. You cascade permissions by setting the Cascade property to True.

Note. The PORTAL_PAGELETS folder is the parent folder for all PageletCategories. Its security is set to public. PeopleSoft does not recommend cascading any permissions from this folder object. Cascade permissions only from the pagelet category (folder).

There are two types of PermissionList collections you can access. One is returned by the Permissions property, the other by the CascadedPermissions property.

Permissions Property

Returns the PermissionValue collection containing the permissions set at the current level, that is, set with that folder or content reference. You can add PermissionValues to this list. Use the Cascade property to cascade the permission you add to child folders and content references.

Note. Folders can contain other folders, but the other objects that use the Permissions property can't contain themselves, that is, content references can't contain other content references, and so on. Therefore, the Cascade property is applicable only to folders, not to any other object.

CascadedPermissions Property

Returns the PermissionValue collection for *all* the permissions for a folder, content reference, PageletCategory object, or Pagelet. It contains any permission list values set in any parent folder.

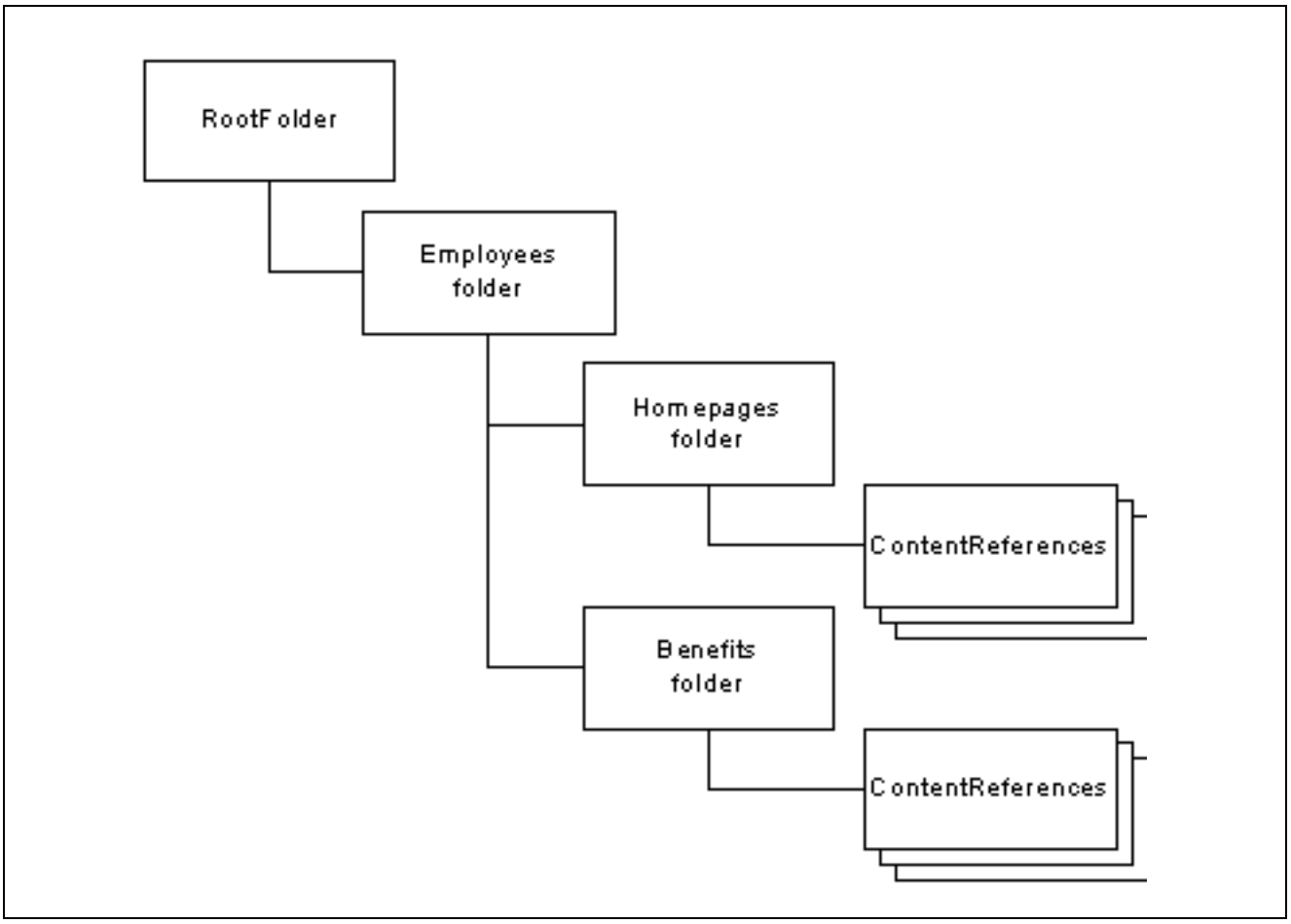
Note. You *cannot* add any PermissionValues to a collection returned by the CascadedPermissions property. You can add values only to the collection returned by the Permissions property.

You can add PermissionValues for a child folder, but you *cannot* delete any of the existing ones that are cascaded. Permissions are augmented, not overwritten, that is, the permissions are the sum of both the parent permissions and whatever is set at the current level.

Therefore, keep the following in mind when working with PermissionValues:

- Set permissions only at the level where they're needed.
- Cascade permissions only when necessary.
- Generally, don't set cascaded PermissionValues for the root folder.
- Only the values in a PermissionValue object cascade. Properties on the folder itself (such as PublicAccess or AuthorAccess) do *not* cascade.

For example, suppose your PortalRegistry had the following hierarchy:



PortalRegistry hierarchy example

The following permission list values are assigned to the PermissionValue objects for the Employees folder, and both of them are cascaded:

- EMPLOYEE
- MANAGER

The Homepages and Benefits folders have the exact same permissions as the Employees folder, that is, all users associated with these two permission lists have access to these folders.

Suppose you decide that you don't want the permission list EMPLOYEE to access the Benefits folder. If you delete the EMPLOYEE PermissionValue from the Benefits PermissionValue collection, you have *not* altered the permissions. The permissions set at the parent folder, that is, at the Employees folder, can't be deleted, they can only be added to.

To make this change, you must:

- Change the Cascaded property for the EMPLOYEES PermissionValue in the Employees folder to False
- Add EMPLOYEES as a PermissionValue object to the Homepages folder.

See Also

Chapter 21, PortalRegistry Classes , Cascade, page 1182

Chapter 21, PortalRegistry Classes , Setting Permissions Using the PermissionValue Object, page 1253

PeopleTools Security, Working With Permission Lists

Working With ValidFrom and ValidTo

Folders, content references, and tab definitions have both ValidFrom and ValidTo dates. What's the difference and how are they used in the PortalRegistry API?

- A ValidFrom date is when something begins.
- A ValidTo date is when something ends.

For example, suppose your HR department has a page describing the benefits for your employees, and that page changed every calendar year. This means the page for the year 2000 has a **ValidFrom** date of 01/01/2000 and a **ValidTo** date of 12/31/2000. The benefits page for the year 2001 has a ValidFrom date of 01/01/2001 and a ValidTo date of 12/31/2001.

Folders, content references, and tab definitions are returned in their collections regardless of the ValidTo and ValidFrom dates. You must take these dates into account and only display to the end-user those values that should be seen.

In addition, a ValidFrom should always come *before* a ValidTo date. If they are set incorrectly, you receive a runtime error.

For all newly created folders, content references, and tab definitions, the default value for both these properties is Null (""), that is, they begin immediately and do not expire.

Doing Error Handling

All errors for the PortalRegistry classes, like the other APIs, are logged in the PSMessages collection, instantiated from a session object.

The PortalRegistry classes log errors that occur with methods immediately, and errors that occur with properties only after a method is executed.

For example, suppose you specified an invalid name when you were trying to delete a folder. The method (DeleteItem) returns False, and the error is logged in the PSMessages collection immediately.

Now suppose you created a new folder, and specified an invalid ValidTo date. The error won't be logged in the PSMessages collection until you tried to save your changes.

When you want to check for errors depends on your application. When users are entering data dynamically, and your program is registering their data in the portal, you may want to check for errors often. If you're using a batch program, you may want to check for errors only after the Open, Save, Insert, and Delete methods.

Most methods return a Boolean value indicating success or failure. After the failure of a method, you may want to check the PSMessages collection to determine the exact error.

```
Local ApiObject &MySession;
Local ApiObject &ErrorCol;
Local ApiObject &FolderCol, &Folder, &Registry;
Local Boolean &Open;/* Access the current session */

&MySession = %Session;
If &MySession Then
   /* connection is good */

&Registry = &MySession.GetRegistry();

&Open = &Registry.Open("CUSTOMER");

If &Open Then
   /* Registry opened successfully */
   /* do processing */
Else
   /* Do error checking */

&ErrorCol = &MySession.PSMessages;
For &I = 1 to &ErrorCol.Count
   /* do processing */
End-For;

   End-If;
Else
   /* do processing for no connection */
End-If;
```

See Also

Chapter 29, Session Class , Error Handling , page 1560

Understanding the Life-Cycle of a PortalRegistry Object

At runtime, there are certain things you want to do with a PortalRegistry object, like getting an instance of it, adding tabs, editing content references, and so on. The following is an overview of this process. These steps are expanded in other sections.

1. Perform one of these steps.
    a. Execute the GetPortalRegistry method on the PeopleSoft session object to get a PortalRegistry object, or
    b. Use the FindPortalRegistries method on a session object to get a collection of all PortalRegistries. Select a PortalRegistry from the collection.

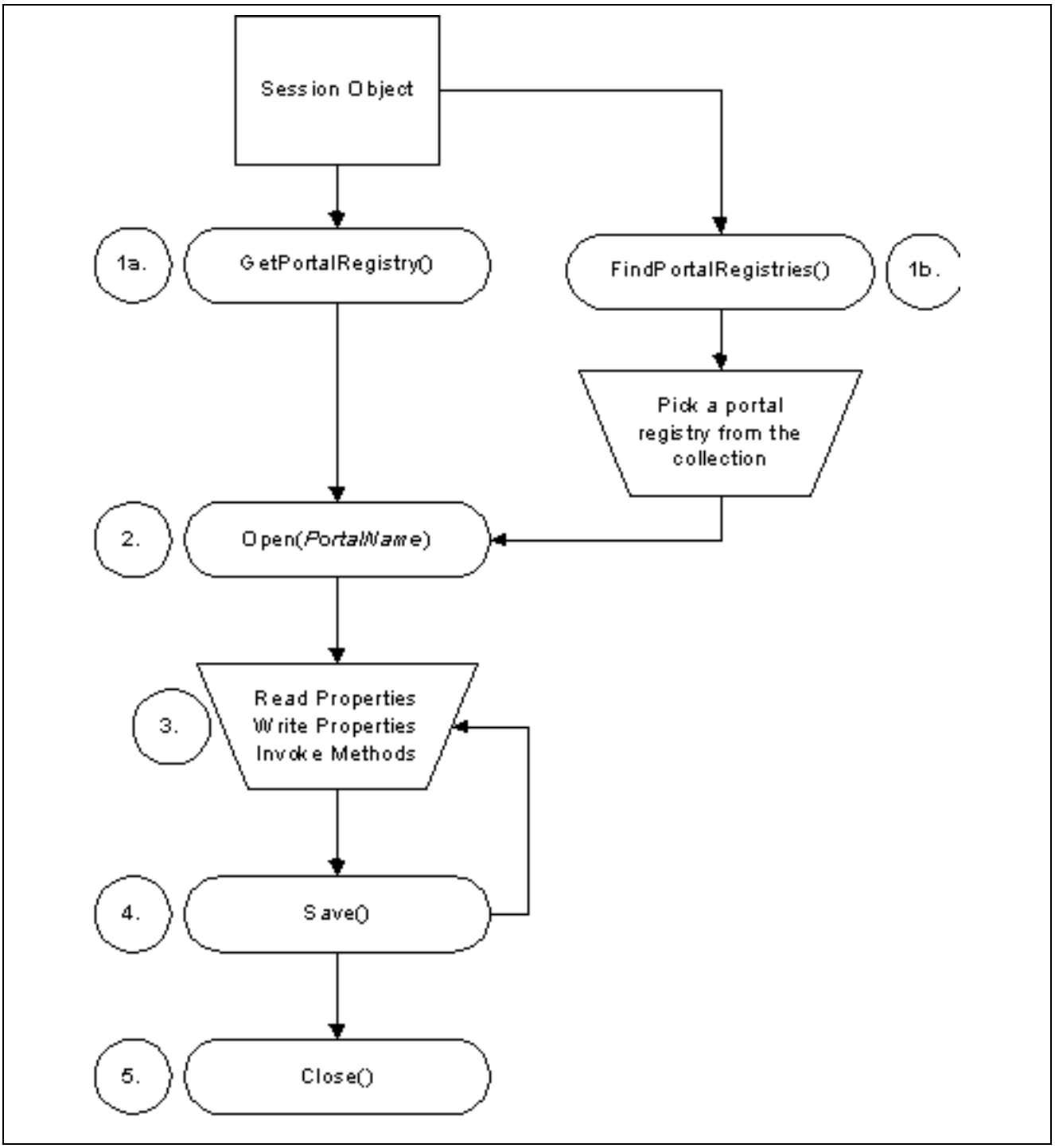
2. Open the PortalRegistry, using the portal name.
3. After you have an open PortalRegistry object, you can do various actions such as:
   - Add content references to the existing folders.
   - Add folders and the content references for them.
   - Add or change Nodes.
   - Modify the PortalRegistry properties.
   - Modify the existing content references, folders, tabs, homepages, and so on.
4. After you make your changes, you must save your work. The Save method must be executed at the appropriate level, such as at the folder level for changes to a folder, at the PortalRegistry level for changes to a node, and so on.

   A property value change isn't committed to the database until the parent object is saved.

   Some methods commit data to the database only when the parent object is saved. However, other methods cause data to be committed to the database as soon as they are executed, like InsertItem on a folder. Methods that automatically make database changes are noted as such in their description.

   See Chapter 21, PortalRegistry Classes , Saving Content Considerations, page 1112.
5. When you finish all operations for a PortalRegistry, close the object.

Note. PeopleSoft recommends that you open and close every PortalRegistry in a single PeopleCode event. You shouldn't open a PortalRegistry object and keep it open across multiple events. You should also keep only one PortalRegistry object open at a time.



Life-Cycle of a PortalRegistry object

Examples of using the PortalRegistry objects are provided at the end of this chapter.

See Also

Chapter 21, PortalRegistry Classes , PortalRegistry Classes Example , page 1242

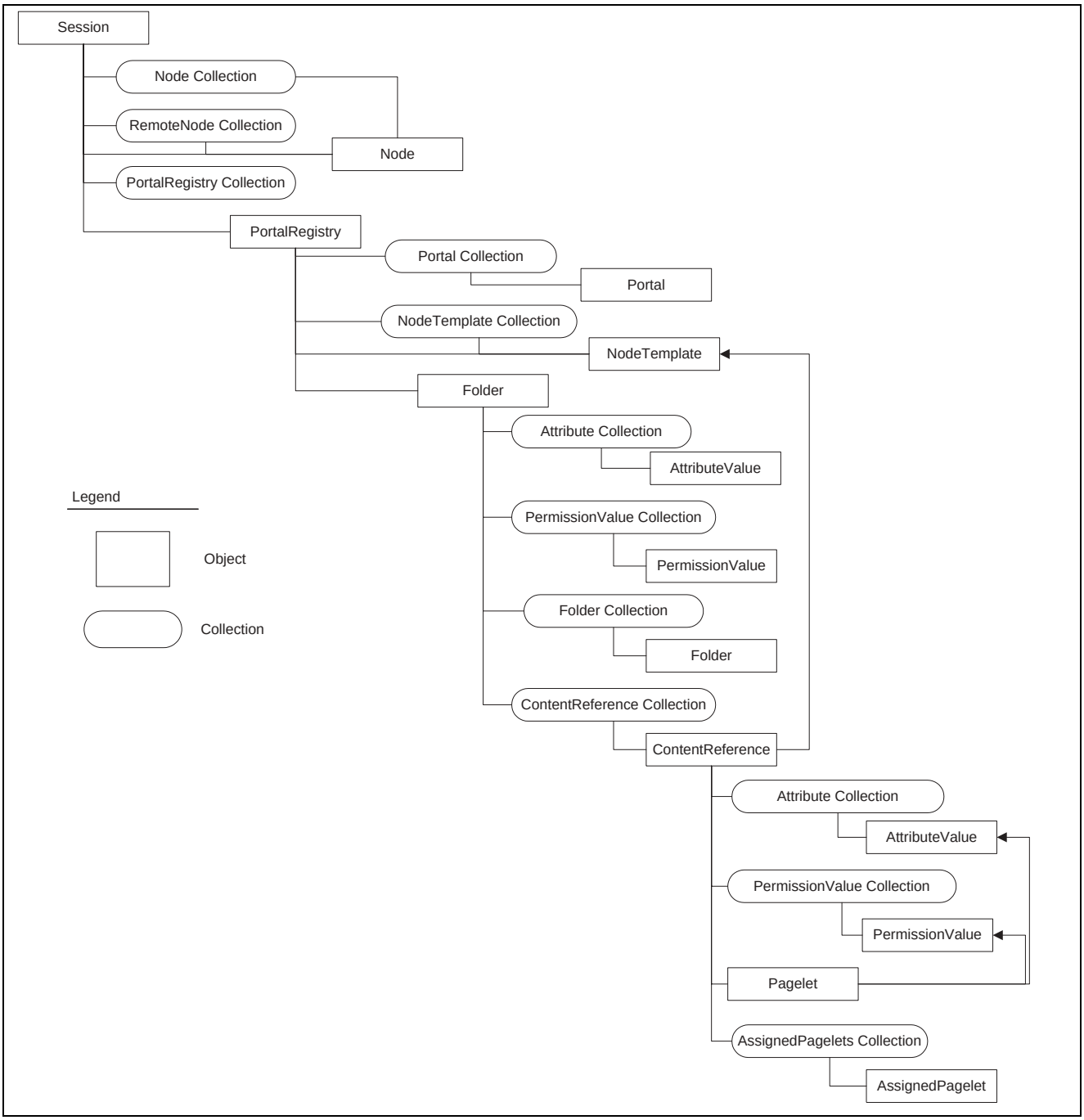
Viewing the PortalRegistry Class Hierarchy

There are many different classes used with the PortalRegistry API. The following flowchart shows the different classes, and how they are interrelated.

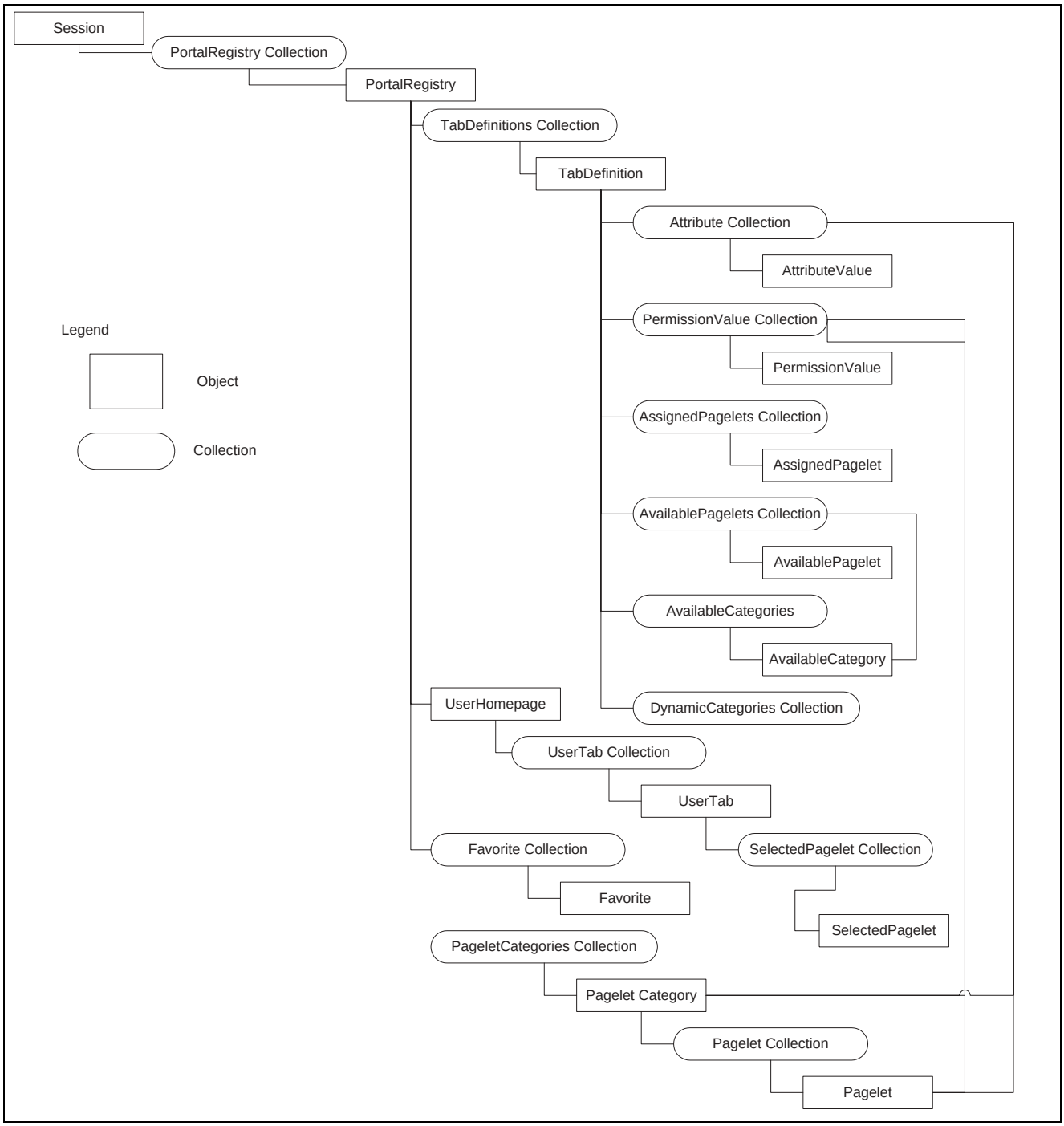
Hierarchy Considerations

The following are considerations for the representation of the hierarchy:

- The flowchart shows only two levels of folders. In reality, you can embed as many levels of folders as you need.
- From a PortalRegistry object, you must access the root folder before you can access the sub-folders for that PortalRegistry.



PortalRegistry class hierarchy (Part 1 of 2)



PortalRegistry class hierarchy (Part 2 of 2)

Using Content References

Content references have a number of properties, but several properties work together to define the type of Content reference. The values of these properties are interdependent, that is, the value of one indicates the values of others. The properties are:

- UsageType
- TemplateType
- URLType
- Nodes and URL

UsageType

This is the primary specifier for the type of content reference. There are a number of different types of content references, but all content references can be categorized into the following major types:

- Target
- Template
- Portal Component

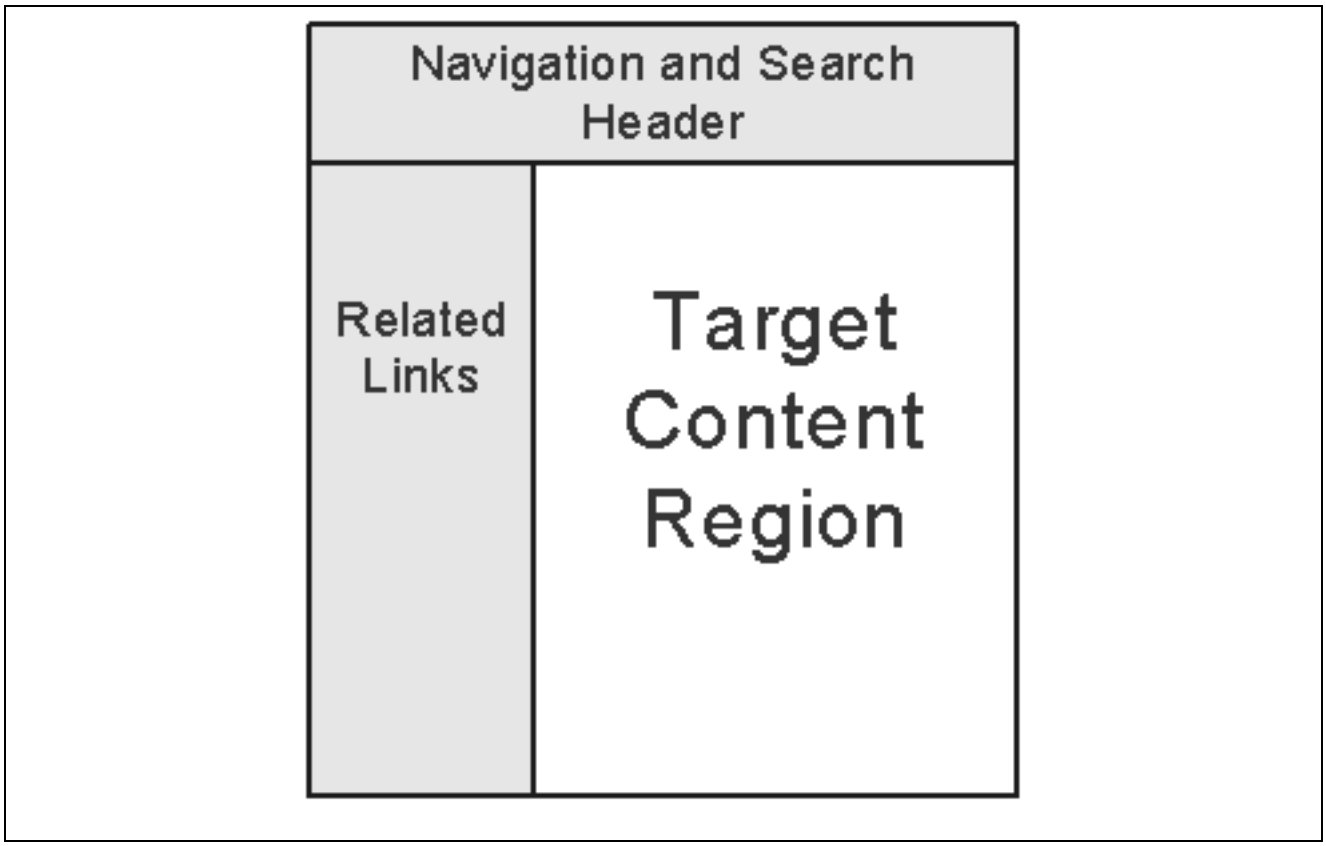
Target

The *target* is the page referenced by a URL in the user's browser. This is the main transaction or page that the user wants, and the portal may place other content around this target page based on the template. The template describes any other content, and where to place it, on the page. The template is either gotten from the content reference or a set of default templates.

Template

A portal *template* defines how the portal creates a user's page. It's an HTML document that describes the content and where the content is placed. The template specifies one target and zero or more portal components.

The portal template in the following example is comprised of four separate template components: one for the navigation and search header, one for related links, one for the target content region, and one for the overall template which specifies where the other components should be placed. At runtime, the target content region would be filled by the HTML returned by the target page, as would the other template component regions.



A Portal template

See *PeopleTools Internet Technology,* Designing Portal Templates.

Portal Component

A *portal component* is an HTML document, or something that produces an HTML document. The portal component must be defined within a template.

UsageType Values

The following table matches the general types of target, template, and portal component to the actual values of the UsageType property.

General Type	UsageType Value
Target	Target (TARG)
Template	Frame template (FRMT) or HTML template (HTMT)
Pagelet	Pagelet (HPGC)
Homepage Tab	Homepage Tab (HPGT)

More specifically:

- A UsageType value of TARG specifies a content reference that is a target.
- A UsageType value of FRMT specifies a content reference that is an HTML frame template.
- A UsageType value of HTMP specifies a content reference that is an HTML template.
- A UsageType value of HPGC specifies a content reference that is a PeopleSoft homepage component (pagelet).
- A UsageType value of HPGT specifies a Homepage tab.

TemplateType

This property is valid only when the UsageType property is a target. For target type content references (TARG) this controls whether the portal looks for and uses a template to wrap the target.

TemplateType	Meaning
NONE	There is no template for this target
HTML	There is some kind of HTML template for this target

URLType

This property gives information about what format the URL is in.

URLType Value	Meaning
UEXT	URL points to a non-PeopleSoft (external) URL
UMPG	URL points to a PeopleSoft mobile page
UPGE	URL points to a component
UPHP	URL points to a homepage tab
UPTM	URL points to a template
USCR	URL points to an iScript
UGEN	URL points to a generic PeopleSoft URL.

The URL property is always required (it's one of the parameters for the InsertItem method.) The format of this parameter (or property) depends on the other properties.

See Also

PeopleTools Internet Technology, Understanding PeopleSoft Portal Technology

Nodes and URL

Nodes and the URL property work together and are interrelated. The node is how to 'register' a logical name for a webserver (the webserver name, on the servlet, and so on) not the actual details. This way, content references don't change when a webserver changes.

For example, suppose you had content that you referenced on the HRMS webserver. However, the machine name for that server changed. You can change the URI of the Node, instead of changing every content reference that referred to that content.

At least one node must be specified as the default local node. Nodes are also required for PeopleSoft components and iScripts.

Considerations When Using Nodes and the URL Property

When a content reference is created it is 'registered' with its URL (the portal, and others, typically find a content reference by its URL). The node is used to create a logical name for the webserver, servlet, and so on, so these details are not included in a content reference's URL. When details of a webserver change, such as at installation time, only the URI for the content provider must change. You don't have to change the URL for any content references.

When the node is specified, the content provider's URI is concatenated with the URL property.

The format of the URL property depends on the content it's pointing to.

Summary

The following table summarizes the interrelations between the different content reference properties.

UsageType	TemplateType	StorageType	URLType
Target (TARG)	HTML	Remote (RMTE)	Component (UPGE)
Target (TARG)	HTML	Remote (RMTE)	Internet Script (USCR)
Target (TARG)	HTML	Remote (RMTE)	External (UEXT)
Target (TARG)	NONE	Remote (RMTE)	Component (UPGE)
Target (TARG)	NONE	Remote (RMTE)	Internet Script (USCR)
Target (TARG)	NONE	Remote (RMTE)	External (UEXT)
Frame template (FRMT), HTMP template	NONE	Remote (RMTE)	Internet Script (USCR)
Frame template (FRMT), HTMP template	NONE	Local (LOCL)	N/A

UsageType	TemplateType	StorageType	URLType
Homepage (HPGT)	NONE	Local (LOCL)	N/A
Template component (TMPC), Homepage component (HPGC)	N/A	Remote (RMTE)	Component (UPGE)
Template component (TMPC), Homepage component (HPGC)	N/A	Remote (RMTE)	Internet Script (USCR)
Template component (TMPC), Homepage component (HPGC)	N/A	Remote (RMTE)	External (UEXT)
Template component (TMPC), Homepage component (HPGC)	N/A	Local (LOCL)	N/A

Naming Conventions

If you create two content references with the exact same URL, the second one fails.

For example, suppose you create an external content reference with a URL of www.peoplesoft.com. Then you create a second content reference that has a node of **PeopleSoft**, whose URI is www.peoplesoft.com. The creation of the second content reference fails because the URL already exists.

If you have multiple nodes with the same URI, FindCRefByURL looks for the specified content reference with all those nodes.

If you have multiple nodes with the same URI, but no registered content references, the system uses the template from the alphabetically first node it finds.

Deleting Content Considerations

Be extremely careful when you delete any content. There may be more than one object relying on the content you delete.

- If you try to delete a template currently used by a content reference, the node template is set as the default template for the portal, you receive an error when you try to save the item.
- If you delete a homepage template for a user, the system tries to use the default user's template first, before resorting to the portal default template.

- If you try to delete a content provider that is currently used by a content reference, you receive an error and cannot save the PortalRegistry object.

Warning! If you delete a folder, you delete *all* content in the folder. If you delete a folder that contains other folders, that is, a parent folder, all the child folders, and all the content references are deleted.If you delete a PortalRegistry, you delete *everything*. Your entire PortalRegistry is gone, all the folders, content references, templates, and so on. Do not delete a PortalRegistry object unless you are absolutely certain you want to.

Saving Content Considerations

The following PortalRegistry classes have Save methods:

- PortalRegistry
- Folder
- Content reference
- TabDefinition
- PageletCategory
- Pagelet
- Homepage
- Favorites collection

This means if you change a folder, you must save the folder. If you change a folder and only save at the PortalRegistry level, your changes are *not* saved.

Some classes do *not* have a Save method. For these classes, you must save the parent object.

- If you change a node, node template, or remote portal, you must use the PortalRegistry Save method.
- If you change an AttributeValue you must use the Save method with the item that contains the AttributeValue (folder, content reference, PageletCategory, or Pagelet).
- If you change a PermissionValue, you must use the Save method with the item that contains the PermissionValue (folder, content reference, PageletCategory or Pagelet).
- If you change a UserTab or a SelectedPagelet, you must use the Homepage Save method.

Data Type of a PortalRegistry Object

PortalRegistry objects, and all objects instantiated from a PortalRegistry object, are declared as data type ApiObject. For example,

```
Local ApiObject &MyRegistry;

Local ApiObject &MyFolder, &MyAttributeValue;
```

Note. PortalRegistry objects can only be declared as Local.

Considerations Using Local Variables

When a local variable has a reference to an object and the end-user clicks the Back button on a browser, the local variable is set to NULL. This is always logged in the trace file.

Considerations Using Global Variables

Global variables are not available to a portal or applications on separate databases. They are available only on applications and Portals in the *same* database.

Scope of a PortalRegistry Object

A PortalRegistry object can be instantiated from the following language environments:

- PeopleCode
- OLE/COM
- C/C++
- Java

See Also

Chapter 21, PortalRegistry Classes , PortalRegistry Classes Example , page 1242

PortalRegistry Reference

The following sections provide more detail of the properties, methods, and other objects that you can use with a PortalRegistry object. If you don't want to programmatically change a portal registry, you can use the Portal Administration Tool pages instead.

See Also

PeopleTools Internet Technology, Understanding PeopleSoft Portal Technology

Session Class Methods

PortalRegistry objects don't have any built-in functions. They are instantiated from a session object. In this section, we discuss the Session Object methods. The methods are listed in alphabetical order.

See Also

Chapter 1, PeopleCode Built-in Functions, GetSession , page 269
Chapter 3, System Variables , %Session , page 577

FindPortalRegistries

Syntax

FindPortalRegistries(*Name*)

Description

The **FindPortalRegistries** method, used with the session object, returns a reference to a PortalRegistry Collection, filled with zero or more PortalRegistry objects matching the *Name* parameter. The *Name* parameter takes a string value.

You can use a partial key to get a smaller subset of the PortalRegistry collection. For example, to get a collection of all the PortalRegistry objects whose names start with the letter "B", specify just the letter B for *Name*:

```
&MyColl = &MySession.FindPortalRegistries("B");
```

Parameters

Parameter	Description
Name	Specify the name of the PortalRegistry object to find. This parameter takes a string value.

Returns

A PortalRegistry Collection object.

Example

The following example returns a collection with references to all PortalRegistry objects.

```
Local ApiObject &MySession;
Local ApiObject &MyColl;

&MySession = %Session;
&MyColl = &MySession.FindPortalRegistries("");
```

GetLocalNode

Syntax

GetLocalNode()

Description

Use the **GetLocalNode** method to return a reference to the node defined as the local node for this session.

Parameters

None.

Returns

A reference to a node object if successful, Null otherwise.

Example

```
&Node = %Session.GetLocalNode();
```

See Also

Chapter 21, PortalRegistry Classes , Node Class, page 1137.

GetNodes

Syntax

```
GetNodes()
```

Description

Use the **GetNodes** method to return a collection of nodes for the session.

Parameters

None.

Returns

A reference to a collection of nodes if successful, Null otherwise.

Example

```
&NodeColl = %Session.GetNodes();
```

See Also

Chapter 21, PortalRegistry Classes , Node Collection, page 1140.

GetPortalRegistry

Syntax

```
GetPortalRegistry()
```

Description

The GetPortalRegistry method returns an empty PortalRegistry object. You can then open or delete an existing PortalRegistry, or create a new one.

Parameters

None.

Returns

An empty PortalRegistry object.

Example

```
Local ApiObject &MyPortal;

&MyPortal = %Session.GetPortalRegistry();
PORTAL_NAME = %Request.GetParameter("PORTAL_NAME");
&Portal.Open(PORTAL_NAME);
```

GetRemoteNodes

Syntax

GetRemoteNodes()

Description

Use the **GetRemoteNodes** method to return a collection of nodes for the session.

Note. This method returns both local and remote nodes.

Parameters

None.

Returns

A reference to a collection of nodes if successful, Null otherwise.

Example

```
&NodeColl = %Session.GetNodes();
```

See Also

Chapter 21, PortalRegistry Classes , RemoteNode Collection, page 1142.

PortalRegistry Class

A PortalRegistry object is returned from:

- The GetPortalRegistry session method.
- The PortalRegistry Collection Methods First, ItemByName, or Next.

Note. In addition to the following methods, the PortalRegistry class has methods used with the Search API.

See Also

Chapter 21, PortalRegistry Classes , GetPortalRegistry, page 1115

Chapter 21, PortalRegistry Classes , PortalRegistry Collection Methods , page 1135

Chapter 21, PortalRegistry Classes , Changing PortalRegistry Properties, page 1242

Chapter 28, Search Classes , page 1509

PortalRegistry Class Methods

In this section, we discuss the PortalRegistry class methods. The methods are discussed in alphabetical order.

Close

Syntax

```
Close()
```

Description

The Close method closes the PortalRegistry object, that is, this method sets the object to the state it was in immediately after the GetPortalRegistry was done on the PeopleSoft Session object. Any unsaved changes are discarded. The **Close** method can be used only on an open PortalRegistry, not a closed one. This means you must have opened the PortalRegistry with the Open method before you can close it.

Parameters

None.

Returns

A Boolean value: True if the PortalRegistry object is successfully closed, False otherwise.

Example

```
&Rslt = &MyRegistry.Close();

If Not &Rslt Then
   /* registry not closed − do error processing */
End-if;
```

See Also

Chapter 21, PortalRegistry Classes , Open, page 1128, Chapter 21, PortalRegistry Classes , Save, page 1131.

CopyObject

Syntax

```
CopyObject(sourcePortalName, sourceRefType, sourceObjName, targetPortalName,
targetPrntFldrName, copyChildren)
```

Description

Use the **CopyObject** method to copy folders and content references between portal registries.

If the object being copied has related HTML, the HTML is also copied and named according to the current portal naming convention for HTML objects, which is:

PR_<*portalname*>_<*objectname*>

where <*portalname*> is up to the first 8 characters of the portal name and <*objectname*> is up to the first 17 characters of the object name.

Considerations Using CopyObject

The source portal name and the target portal name cannot be the same name.

The *targetPrntFldrName* must be a folder that exists in the portal identified by the target portal name. It cannot be null (or empty). If you need to copy an entire portal, the portal must first be "created" using the Create method, before copying over all the folders and content references.

The *copyChildren* parameter is ignored for content references. However, it still must be set to "Yes" or "No".

Parameters

Parameter	Description
sourcePortalName	Specify the name of the portal from which the object is to be copied.
sourceRefType	Specify the type of object to be copied. Values are: • "C" for content references • "F" for folders This parameter is case-sensitive.
sourceObjName	Specify the name of the object to be copied.
targetPortalName	Specify the name of the portal to which the object will be copied.
targetPrntFldrName	Specify the name of the folder that will be the parent folder for the source object.
copyChildren	Specify whether to also copy the children of the source object. This parameter is valid only with folder objects. Values are: • "Yes" copy child objects • "No" do not copy child objects This parameter is case-sensitive.

Returns

A Boolean value: True if object copied successfully, False otherwise.

Example

```
&Success = &Portal.CopyObject("PORTAL", "F", "PEOPLETOOLS_QUALITY", "BOGUS",⇒
 "PORTAL_ROOT_OBJECT", "Yes");
If (&Success = False) Then
   WinMessage("portal copy failed");
```

```
   Exit;
End-If;
```

Create

Syntax

Create(*RegistryName*)

Description

The **Create** method creates a new PortalRegistry in the PortalRegistry object called *RegistryName*. The specified registry must be a *new* registry. The Create method returns False if the registry already exists.

The new PortalRegistry is immediately committed to the database. If you change any property of a PortalRegistry after you create the object, use the Save method to commit your changes to the database.

When a registry is created, a folder called *Root* is automatically created. This is the root folder for the registry.

Note. If you're using Visual Basic, you must check that the PortalRegistry is actually created. If you use a duplicate name, a zero is returned, but no error results.

Parameters

Parameter	Description
RegistryName	The name of the registry to create. This parameter takes a string value. If you specify a registry that already exists, this method returns a False value.

Returns

A Boolean value: True if the PortalRegistry object is successfully created, False otherwise.

Example

```
&PORTAL_NAME = MY_PORTAL_RECORD.PORTALNAME;
&MyPortal = %Session.GetPortalRegistry();

If NOT &MyPortal.Create(&PORTAL_NAME) Then;
   /* portal not created – do error processing */
End-If;
```

See Also

Chapter 21, PortalRegistry Classes , Open, page 1128, Chapter 21, PortalRegistry Classes , Save, page 1131, Chapter 21, PortalRegistry Classes , Close, page 1117, Chapter 21, PortalRegistry Classes , Delete, page 1120.

CreateRemote

Syntax

CreateRemote(*PortalName, RemoteNodeName*)

Description

Use the **CreateRemote** method to create a remote portal. Although you can have more than one portal on a node, they must all be uniquely named.

Parameters

Parameter	Description
PortalName	Specify the name of the new portal to create on the remote node.
RemoteNodeName	Specify the name of the remote node to create the new portal on.

Returns

A Boolean value: True if the PortalRegistry object is successfully created, False otherwise.

See Also

Chapter 21, PortalRegistry Classes , Create, page 1119, Chapter 21, PortalRegistry Classes , Portals, page 1134.

Delete

Syntax

Delete(*RegistryName*)

Description

The **Delete** method deletes the PortalRegistry *from the database,* including any data and tables.

Warning! If you delete a PortalRegistry, you delete *everything.* Your entire PortalRegistry is gone, all the folders, content references, templates, and so on. Do not delete a PortalRegistry object unless you are absolutely certain that you want to.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for deletion, then actually deleted later. The item is deleted from the database *as soon as* the method is executed.

The **Delete** method can only be used with a *closed* registry, it cannot be used on an open registry. Before you use the **Delete** method, you must explicitly close the PortalRegistry object (with the Close method.) The Delete method returns False if you try to delete an open PortalRegistry object.

Parameters

Parameter	Description
RegistryName	The name of the registry to delete. This parameter takes a string value. If you specify a registry that doesn't exist, this method returns a False value.

Returns

A Boolean value: True if the PortalRegistry object is successfully deleted, False otherwise.

Example

The following example deletes all PortalRegistry objects that start with Customer.

```
Local ApiObject &MySession;
Local ApiObject &MyPortal;

&MySession = %Session;

&MyPortal = &MySession.GetPortalRegistry();

&MyPortal.Delete("HRMS_99");
```

See Also

Chapter 21, PortalRegistry Classes , Close, page 1117, Chapter 21, PortalRegistry Classes , Open, page 1128.

DeleteHomepage

Syntax

DeleteHomepage()

Description

The **DeleteHomepage** method deletes the current user's homepage. This method is valid only after a user has opened a PortalRegistry object.

Parameters

None.

Returns

A Boolean value: True if the homepage is successfully deleted, False otherwise.

See Also

Chapter 21, PortalRegistry Classes , UserHomepage Class, page 1227.

FindCRefByName

Syntax

FindCRefByName(*Name*)

Description

The **FindCRefByName** method returns the content reference object corresponding to *Name*. The name is a unique identifier for each content reference.

Considerations on Returned Content References

This method returns content reference objects that aren't yet valid as well as ones that are no longer valid. When you create your program, you must check for these properties (ValidTo and ValidFrom) if you don't want to use them.

This method returns content reference objects that you aren't authorized to. When you create your program, you should always check the Authorized property. This is the only property you can view from an object that you're not authorized to view.

Parameters

Parameter	Description
Name	A unique name within the registry that identifies the content reference. This parameter takes a string value. This parameter uses the name, not the label, of a content reference.

Returns

What this method returns depends on the condition of the content reference:

- If the content reference name is valid and the end-user has access to the content reference, a content reference object is returned.
- If the content reference is valid but the end-user doesn't have access to it a content reference object is returned, however, the only property you can access is the Authorized property.
- If you specified an invalid name this method returns NULL.

Example

The following example returns a content reference object:

```
&CRef = &Portal.FindCRefByName("GLOBAL_PAYROLL");
```

See Also

Chapter 21, PortalRegistry Classes , Content Reference Class, page 1160.

FindCRefByURL

Syntax

FindCRefByURL(*URL*)

Description

The **FindCRefByURL** method returns the content reference object corresponding to *URL*. The URL specified by *URL* must be an absolute URL.

Note. The portal registry API needs the current URI of the local node to work. During runtime, it gets this information from the current webserver. For Application Engine programs, there isn't a current webserver, so it has to get this information from the database.

If you have multiple content providers with the same URI, FindCRefByURL looks for the specified content reference with all those content providers.

To access pagelet categories and their associated pagelets, use the PageletCategories collection from the PortalRegistry object, not FindCRefByURL.

Considerations on Returned Content References

This method returns content reference objects that aren't yet valid as well as ones that are no longer valid. When you create your program, check for these properties (ValidTo and ValidFrom) if you don't want to use them.

This method returns content reference objects that the end-user isn't authorized to. When you create your program, always check the Authorized property. This is the only property you can view from an object that the end-user isn't authorized to view.

Parameters

Parameter	Description
URL	A URL that represents the content. This parameter takes a string value. This URL must be an absolute URL. This parameter is case-insensitive.

Returns

What this method returns depends on the condition of the content reference:

- If the end-user has access to the URL, a content reference object is returned.
- If the content reference is registered, but the end-user doesn't have access to it, a content reference is returned, but the only property you can access is the Authorized property.
- If a URL isn't registered or is invalid, this method returns NULL.

Example

The following example finds a content reference from a URL:

```
&UserCRef = &Portal.FindCRefByURL("http://www.PeopleSoft.Com");
```

See Also

Chapter 21, PortalRegistry Classes , GetQualifiedURL, page 1126, Chapter 21, PortalRegistry Classes , QualifiedURL, page 1167, Chapter 21, PortalRegistry Classes , Content Reference Class, page 1160.

FindFolderByName

Syntax

FindFolderByName(*Name*)

Description

The **FindFolderByName** method returns the Folder object corresponding to *Name*. The name is a unique identifier for each folder.

Considerations on Returned Folders

This method returns Folder objects that aren't yet valid as well as ones that are no longer valid. When you create your program, check for these properties (ValidTo and ValidFrom) if you don't want to use them.

This method returns folder objects that you aren't authorized to. When you create your program, always check the Authorized property. This is the only property you can view from an object that you're not authorized to view.

Parameters

Parameter	Description
Name	A unique name within the registry that identifies the folder. This parameter takes a string value. This parameter takes the name of a folder, not the label.

Returns

What this method returns depends on the condition of the folder:

- If the folder name is valid and the end-user has access to the folder, a folder object is returned.
- If the folder is valid but the end-user doesn't have access to it a folder object is returned, however, the only property you can access is the Authorized property.
- If you specified an invalid name this method returns NULL.

Example

The following example returns a folder named ROOT:

```
&MyFolder = &MyPortal.FindFolderByName("ROOT");
```

The following example returns the folder object for an already instantiated content reference:

```
&Folder = &Portal.FindFolderByName(&CRef.ParentName);
```

See Also

Chapter 21, PortalRegistry Classes , Folder Class, page 1151.

FindPgltByName

Syntax

FindPgltByName(*PageletName*)

Description

The **FindPgltByName** method returns the pagelet object corresponding to *PageletName*. The name is a unique identifier for each pagelet.

Considerations on Returned Pagelets

This method returns pagelet objects that aren't yet valid as well as ones that are no longer valid. When you create your program, check for these properties (ValidTo and ValidFrom) if you don't want to use them.

This method returns pagelet objects that you aren't authorized to. When you use create your program, always check the Authorized property. This is the only property you can view from an object that you're not authorized to view.

Parameters

Parameter	Description
PageletName	A unique name within the registry that identifies the Pagelet. This parameter takes a string value. This parameter takes the name of a Pagelet, not the label.

Returns

What this method returns depends on the condition of the Pagelet:

- If the Pagelet name is valid and the end-user has access to the Pagelet, a Pagelet object is returned.
- If the Pagelet is valid but the end-user doesn't have access to it a Pagelet object is returned, however, the only property you can access is the Authorized property.
- If you specified an invalid name this method returns NULL.

Example

The following example returns a pagelet named HomePg_Dictionary:

```
&MyPglt = &MyPortal.FindPgltByName("HomePg_Dictionary");
```

See Also

Chapter 21, PortalRegistry Classes , Pagelet Class, page 1217.

GetAbsoluteContentURL

Syntax

GetAbsoluteContentURL(**Node**.*NodeName*, *URL*)

Description

The **GetAbsoluteContentURL** method returns the absolute unwrapped simple URL of the content, in the context of the current portal. For example, if the PortalRegistry object accessed a portal called Employees, and the method were called like this:

```
&Registry.getAbsoluteContentURL(Node.CRM,  "/c/SERVICES.ORDERS.GBL");
```

It would return the following string:

```
http://crmserver/servlets/psc/crmHome/Employees/CRM/c/SERVICES.ORDERS.GBL
```

In the returned string, the portion of the string from server name through *PS_Home* (crmHome in the example) are the ones associated with the content node (CRM in the example). The portal is the current portal, and the rest of the URL is the URL string passed in.

Parameters

Parameter	Description
NodeName	Specify the name of the node that contains the content, prefixed with the reserved word **NODE**. You can also use a string, such as %Node, for this value.
URL	Specify the relative URL pointing to the content that you want the absolute URL generated for.

Returns

A string containing the absolute URL.

GetDefaultHPTabOID

Syntax

```
GetDefaultHPTabOID()
```

Description

Use the **GetDefaultHPTabOID** method to return the name of the first homepage tab that is found and authorized for the current UserID.

Parameters

None.

Returns

A string

GetQualifiedURL

Syntax

```
GetQualifiedURL(ContentProvider, RelativeURL)
```

Description

Note. This method is maintained only for backward compatibility. If your existing code uses this method, it actually returns the value from **GetAbsoluteContentURL** method. New applications should use the GetAbsoluteContentURL method.

See Also

Chapter 21, PortalRegistry Classes , GetAbsoluteContentURL, page 1125.

GrantPermissionForComponent

Syntax

```
GrantPermissionForComponent(MenuName, ComponentName, Market, PermListName, NodeName)
```

Description

Use the **GrantPermissionForComponent** method to grant the specified permission to the specified component. In addition, the specified permission is granted to any component references that point to the specified component, and any parent folders.

If you use the string "LOCAL_NODE" as *NodeName*, the system uses the node name currently defined as local.

Parameters

Parameter	Description
MenuName	Specify the menu name, as a string, that the component you want to grant permissions for is associated with.
ComponentName	Specify the component name, as a string, that you want to grant permissions for.
Market	Specify the market, as a string, associated with the component that you want to grant permissions for.
PermListName	Specify the name of the permission list, as a string, that you want to use.
NodeName	Specify the node of the component reference, as a string, that points to the component reference that you want to grant permissions for. If you use the string "LOCAL_NODE", the system uses the node name currently defined as local.

Returns

A Boolean value: True if method completed successfully, False otherwise.

Example

The My Profile Component Reference points to the USERMAINT_SELF component. Using the following example, "ALLPANLS" permission is granted to the MY_PROFILE component reference, as well as "MY INFO", the parent folder..

```
&Portal.GrantPermissionForComponent("MAINTAIN_SECURITY", "USERMAINT_SELF", "GBL",⇒
 "ALLPANLS", "LOCAL_NODE");
```

See Also

Chapter 21, PortalRegistry Classes , RevokePermissionForComponent, page 1130, *PeopleTools Security*, Working With Permission Lists.

GrantPermissionForScript

Syntax

GrantPermissionForScript(*RecordName, FieldName, EventName, FuncName, PermListName, NodeName*)

Description

Use the **GrantPermissionForScript** method to grant the specified permission to the specified iScript. In addition, the specified permission is granted to any component references that point to the specified iScript, and any parent folders.

If you use the string "LOCAL_NODE" as *NodeName*, the system uses the node name currently defined as local.

Parameters

Parameter	Description
RecordName	Specify the record name, as a string, of the record containing the iScript. All iScripts are contained on records whose names start with "WEBLIB".
FieldName	Specify the name of the field, as a string, containing the iScript.
EventName	Specify the name of the event, as a string, containing the iScript. Generally iScripts are contained in the FieldFormula event.
FuncName	Specify the name of the function, as a string, containing the iScript.
PermListName	Specify the name of the permission list, as a string, that you want to use.
NodeName	Specify the node of the component reference, as a string, that points to the component reference that you want to grant permissions for. If you use the string "LOCAL_NODE", the system uses the node name currently defined as local.

Returns

A Boolean value: True if method completed successfully, False otherwise.

Example

```
&Portal.GrantPermissionForScript("WEBLIB_ALERT", "ALERTCOUNT", "FieldFormula",⇒
 "Connect_Alert", "ALLPGS", "Local_Node");
```

See Also

Chapter 21, PortalRegistry Classes , RevokePermissionForComponent, page 1130, *PeopleTools Security*, Working With Permission Lists.

Open

Syntax

Open(*RegistryName*)

Description

The **Open** method opens the PortalRegistry specified by the parameters. The registry must already exist. The **Open** method can be used only with a *closed* PortalRegistry, it cannot be used on an open registry.

Parameters

Parameter	Description
RegistryName	The name of the registry to open. This parameter takes a string value. If you specify a registry that doesn't exist, this method returns a False value.

Returns

A Boolean value: True if the PortalRegistry object is successfully opened, False otherwise.

Example

In the following example, the name of the portal is stored as the value of a field in a record.

```
&PORTAL_NAME = EO_PE_REG_AET.PORTAL_NAME;
&Portal = %Session.GetPortalRegistry();

&Portal.Open(&PORTAL_NAME);
```

See Also

Chapter 21, PortalRegistry Classes , Open, page 1128, Chapter 21, PortalRegistry Classes , Save, page 1131, Chapter 21, PortalRegistry Classes , Close, page 1117, Chapter 21, PortalRegistry Classes , Delete, page 1120.

PermissionListDelete

Syntax

PermissionListDelete(*PermListName*)

Description

Use the **PermissionListDelete** method to delete the specified permission list from the PortalRegistry.

If you try to delete a permission list that is still in use, you receive an error when you try to save the object.

Parameters

Parameter	Description
PermListName	Specify the permission list to delete from the PortalRegistry.

Returns

A Boolean value: True if the permission list is successfully deleted, False otherwise.

PermissionListSaveAs

Syntax

PermissionListSaveAs(*PermListSourceName, PermListTargetName*)

Description

Use the **PermissionListSaveAs** method to copy the specified permission list in the PortalRegistry.

Parameters

Parameter	Description
PermListSourceName	Specify the name of the PermissionList to copy.
PermListTargetName	Specify the name that you want to give the new PermissionList.

Returns

A Boolean value: True if the permission list is successfully copied, False otherwise.

RevokePermissionForComponent

Syntax

RevokePermissionForComponent(*MenuName, ComponentName, Market, PermListName, NodeName*)

Description

Use the **RevokePermissionForComponent** method to revoke the specified permission to the specified component. In addition, the specified permission is revoked for any component references that point to the specified component.

If you use the string "LOCAL_NODE" as *NodeName*, the system uses the node name currently defined as local.

Parameters

Parameter	Description
MenuName	Specify the menu name, as a string, that the component that you want to revoke permissions for is associated with.
ComponentName	Specify the component name, as a string, that you want to revoke permissions for.
Market	Specify the market, as a string, associated with the component that you want to revoke permissions for.
PermListName	Specify the name of the permission list, as a string, that you want to use.
NodeName	Specify the node of the component reference, as a string, that points to the component reference that you want to revoke permissions for. If you use the string "LOCAL_NODE", the system uses the node name currently defined as local.

Returns

A Boolean value: True if method completed successfully, False otherwise.

Example

The My Profile Component Reference points to the USERMAINT_SELF component. Using the following example, "ALLPANLS" permission is revoked to the MY_PROFILE component reference.

```
&Portal.RevokePermissionForComponent("MAINTAIN_SECURITY", "USERMAINT_SELF", "GBL",⇒
 "ALLPANLS", "LOCAL_NODE");
```

See Also

Chapter 21, PortalRegistry Classes , GrantPermissionForComponent, page 1126, *PeopleTools Security*, Working With Permission Lists.

RevokePermissionForScript

Syntax

RevokePermissionForScript(*RecordName, FieldName, EventName, FuncName, PermListName, NodeName*)

Description

Use the **RevokePermissionForScript** method to revoke the specified permission to the specified iScript. In addition, the specified permission is revoked for any component references that point to the specified iScript.

If you use the string "LOCAL_NODE" as *NodeName*, the system uses the node name currently defined as local.

Parameters

Parameter	Description
RecordName	Specify the record name, as a string, of the record containing the iScript. All iScripts are contained on records whose names start with "WEBLIB".
FieldName	Specify the name of the field, as a string, containing the iScript.
EventName	Specify the name of the event, as a string, containing the iScript.
FuncName	Specify the name of the function, as a string, containing the iScript. Generally iScripts are contained in the FieldFormula event.
PermListName	Specify the name of the permission list, as a string, that you want to use.
NodeName	Specify the node of the component reference, as a string, that points to the component reference that you want to revoke permissions for. If you use the string "LOCAL_NODE", the system uses the node name currently defined as local.

Returns

A Boolean value: True if method completed successfully, False otherwise.

Example

```
&Portal.RevokePermissionForScript("WEBLIB_ALERT", "ALERTCOUNT", "FieldFormula",⇒
 "Connect_Alert", "ALLPGS", "Local_Node");
```

See Also

Chapter 21, PortalRegistry Classes , GrantPermissionForScript, page 1127, *PeopleTools Security*, Working With Permission Lists.

Save

Syntax

Save()

Description

The **Save** method saves changes that you made to the PortalRegistry or to a ContentProvider. It does *not* save any changes that you made to a folder, content reference, and so on.

Note. To save changes for a folder or content reference use the save method associated with that object.

Parameters

None.

Returns

A Boolean value: True if the PortalRegistry object is successfully saved, False otherwise.

Example

```
If Not(&MyPortal.Save()) Then
   /* do error checking */
End-if:
```

See Also

Chapter 21, PortalRegistry Classes , Save, page 1151 folder method
Chapter 21, PortalRegistry Classes , Save, page 1161 content reference method

PortalRegistry Class Properties

In this section, we discuss the PortalRegistry class properties. The properties are discussed in alphabetical order.

DefaultTemplate

Description

This property returns or sets the name of the default template for this PortalRegistry object as a string.

If you delete a template for a content reference, and none of the other content references on a page have a template, the default template specified with the Node is used. If there's no template for the Node, the template specified with this property is used.

To return a reference to the content reference that contains the template specified by this property, use the TemplateObject property.

Note. If you change the Default Template for a portal, the template won't take effect until you close all the existing browser windows with the session and open a new browser window.

This property takes only the first 30 characters of a value. If you specify a value longer than 30 characters, the remaining characters are ignored.

This property is read-write.

See Also

Chapter 21, PortalRegistry Classes , TemplateObject, page 1135.

Description

Description

This property returns or sets the description of this PortalRegistry object as a string

The length of this property is 256 characters.

This property is read-write.

Favorites

Description

This property returns a reference to the Favorites Collection for the current end-user.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Favorite Collection, page 1239.

Homepage

Description

This property returns a reference to the Homepage for the current end-user.

This property is read-only.

Name

Description

This property returns the name of the PortalRegistry object as a string.

The length of this property is 30 characters.

This property is read-only.

NodeTemplates

Description

This property returns a reference to a NodeTemplate Collection. This property can be used with a *closed* portal registry, that is, before you open it with the **Open** method.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , NodeTemplate Collection, page 1148.

OwnerId

Description

This property returns or sets the owner ID of the PortalRegistry object as a string.

This property is read-write.

PageletCategories

Description

This property returns a reference to a PageletCategories Collection that contains all the PageletCategories for a portal.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , PageletCategory Collection, page 1214.

Portals

Description

This property returns a reference to all the Portals associated with the PortalRegistry.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Portal Collection, page 1145.

RootFolder

Description

This property returns the root folder object for this PortalRegistry object.

This property is read-only.

TabDefinitions

Description

This property returns reference to a TabDefinitions Collection that contains all the TabDefinitions for a portal.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , TabDefinition Collection, page 1192.

TemplateObject

Description

This property returns a reference to a content reference object that contains the template specified by the DefaultTemplate property. If no template is specified with DefaultTemplate, this property returns Null.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , DefaultTemplate, page 1132.

PortalRegistry Collection

A PortalRegistry Collection is returned by the FindPortalRegistries session class method.

See Chapter 21, PortalRegistry Classes , FindPortalRegistries, page 1114.

PortalRegistry Collection Methods

In this section, we discuss the PortalRegistry collection methods. The methods are discussed in alphabetical order.

First

Syntax

```
First()
```

Description

The **First** method returns the first PortalRegistry object in the PortalRegistry collection.

Example

```
&MyRegistry = &MyCollection.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns the PortalRegistry object with the position in the PortalRegistry collection specified by *number*.

Parameters

Parameter	Description
number	Specify the position number in the collection of the PortalRegistry object that you want returned.

Returns

A PortalRegistry object if successful, NULL otherwise.

Example

```
For &I = 1 to &MyCollection.Count
   &MyRegistry = &MyCollection.Item(&I);
   /* Do processing */
End-For;
```

Next

Syntax

Next()

Description

The **Next** method returns the next PortalRegistry object in the PortalRegistry collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MyRegistry = &MyCollection.Next();
```

PortalRegistry Collection Property

This section discusses the Count property.

Count

Description

This property returns the number of PortalRegistry objects in the PortalRegistry collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Node Class

The node is used by the portal services to determine the URI of the nodes' content. This URI points to the PeopleSoft content service if the node is a PeopleSoft node, and any other URI for an external node.

Use the IsLocal or IsDefault to determine if a node from a RemoteNode collection is remote or local.

Node objects are instantiated from the following:

- From a Session object with the GetLocalNode method.
- From a Node Collection with the First, InsertItem, ItemByName, and Next methods.
- From a RemoteNode Collection with the First, InsertItem, ItemByName, and Next methods.

See Chapter 21, PortalRegistry Classes , GetLocalNode, page 1114; Chapter 21, PortalRegistry Classes , Node Collection, page 1140 and Chapter 21, PortalRegistry Classes , RemoteNode Collection, page 1142.

Node Class Properties

In this section, we discuss the Node class properties. The properties are discussed in alphabetical order.

ActiveNode

Description

This property indicates whether the node has been specified as an active node. This property returns a Boolean value: True, the node is active, False otherwise.

This property is read-only.

AppsRelease

Description

This property returns the release of the PeopleSoft Applications hosted on this node as a string. This property is valid only when the NodeType property is set to PeopleSoft (PIA).

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , NodeType, page 1139.

ContentURI

Description

This property returns the URI for the content webserver of this node as a string.

This property is read-only.

DefaultPortalName

Description

This property returns the name of the portal associated with the node as the default portal for the node.

This property is read-only.

Description

Description

This property returns the description for this node as a string.

The length of this property depends on your system database limit for LONG fields.

This property is read-only.

IsDefault

Description

This property indicates whether the node has been specified as the default local node. This property takes a Boolean value: True, this node has been specified as the default local node, False otherwise.

This property is valid only when the NodeType property is set to PeopleSoft (PIA).

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , NodeType, page 1139.

IsLocal

Description

This property indicates whether this node has been specified as the local PeopleSoft node. This property takes a Boolean value: True, this node has been specified as a local node, False otherwise.

This property is valid only when the NodeType property is set to PeopleSoft (PIA).

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , NodeType, page 1139.

Name

Description

This property returns the name for this node object as a string. The name is a unique identifier for each node object.

Every node name must be unique in the database.

This property is read-only.

NodePassword

Description

This property returns the password for this node object as a string.

This property is read-only.

NodeType

Description

This property indicates what the node is used for. Nodes can either define PeopleSoft or external systems. PeopleSoft nodes have more capabilities than external nodes, therefore several other properties depend on this property.

Values are:

Value	Description
PIA	PIA (Peoplesoft 8.4 simple URL format)
EX	External Node
ICT	ICType (used for 8.1x content)

PIA is the default value on a new Node.

This property is read-only.

PortalURI

Description

This property returns the URI for the portal webserver of this node as a string. This property is valid only when the NodeType property is set to PeopleSoft (PIA).

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , NodeType, page 1139.

ToolsRelease

Description

This property returns the release of PeopleTools running on this node as a string. This property is valid only when the NodeType property is set to PeopleSoft (PIA).

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , NodeType, page 1139.

Node Collection

A node collection contains a set of references to all the local nodes defined in the database. This collection does *not* contain references to remote nodes.

A remote node collection is instantiated by the GetNodes session method.

See Chapter 21, PortalRegistry Classes , GetNodes, page 1115.

Node Collection Methods

In this section, we discuss the Node collection methods. The methods are discussed in alphabetical order.

First

Syntax

First()

Description

The **First** method returns the first Node object in the Node collection.

Parameters

None.

Returns

A Node object.

Example

```
&MyNode = &MyCollection.First();
```

ItemByName

Syntax

ItemByName(*NodeName*)

Description

The **ItemByName** method returns the Node object with the name *NodeName*.

Parameters

Parameter	Description
NodeName	Specify the name of an existing node in the Node collection. If you specify an invalid name, the object returned is NULL.

Returns

A Node object if successful, NULL otherwise.

Example

```
&MyOldNode = &MyPortal.Nodes.ItemByName("HRMS");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next Node object in the Node collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A Node object.

Example

```
&My*Node* = &MyCollection.Next();
```

Node Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of Node objects in the Node Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

RemoteNode Collection

A remote node collection contains a set of references to all the nodes defined in the database, both local and remote.

A remote node collection is instantiated by the GetRemoteNodes session method.

See Chapter 21, PortalRegistry Classes , GetRemoteNodes, page 1116.

RemoteNode Collection Methods

In this section, we discuss the RemoteNode collection methods. The methods are discussed in alphabetical order.

First

Syntax

First()

Description

The **First** method returns the first Node object in the RemoteNode collection.

Parameters

None.

Returns

A Node object.

Example

```
&MyNode = &MyCollection.First();
```

ItemByName

Syntax

ItemByName(*NodeName*)

Description

The **ItemByName** method returns the Node object with the name *NodeName*.

Parameters

Parameter	Description
NodeName	Specify the name of an existing node in the RemoteNode collection. If you specify an invalid name, the object returned is NULL.

Returns

A Node object if successful, NULL otherwise.

Example

```
&MyOldNode = &MyPortal.RemoteNodes.ItemByName("HRMS");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next Node object in the RemoteNode collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A Node object.

Example

```
&My<i>Node</i> = &MyCollection.Next();
```

RemoteNode Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of Node objects in the RemoteNode Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Portal Class

The portal object provides access to portals declared in the session.

A portal object is instantiated by the First, InsertItem, ItemByName, and Next Portal Collection methods

See Chapter 21, PortalRegistry Classes , Portal Collection, page 1145.

Portal Class Method

In this section, we discuss the Portal class methods. The methods are discussed in alphabetical order.

Save

Syntax

Save()

Description

Use the **Save** method to save changes you made to the portal object.

Parameters

None.

Returns

A Boolean value: True if the portal object is successfully saved, False otherwise.

See Also

Chapter 21, PortalRegistry Classes , Save, page 1131 PortalRegistry method

Portal Class Properties

In this section, we discuss the Portal class properties. The properties are discussed in alphabetical order.

HostNodeName

Description

This property sets or returns the name of the node hosting the portal as a string.

This property is read-write.

IsLocal

Description

This property indicates whether the portal is defined as local or remote. This property returns a Boolean value: True, the portal is defined as local, False otherwise.

This property is read-only.

Name

Description

This property returns the name of this portal as a string. This name exactly matches the portal name of the portal registry defined in the database.

This property is read-only.

Portal Collection

The portal collection contains a set of references to each portal defined in the database.

A portal collection is instantiated by the Portals PortalRegistry property.

See Chapter 21, PortalRegistry Classes , Portals, page 1134.

Portal Collection Methods

In this section, we discuss the Portal collection methods. The methods are discussed in alphabetical order.

First

Syntax

```
First()
```

Description

The **First** method returns the first Portal object in the Portal collection.

Parameters

None.

Returns

A Portal object.

Example

```
&MyPortal = &MyCollection.First();
```

ItemByName

Syntax

ItemByName(*PortalName*)

Description

The **ItemByName** method returns the Portal object with the name *PortalName*.

Parameters

Parameter	Description
PortalName	Specify the name of an existing portal within the collection. If you specify an invalid name, the object returns NULL.

Returns

A Portal object if successful, NULL otherwise.

Example

```
&MyPortal = &MyPortal.Portals.ItemByName("HRMS");
```

Next

Syntax

Next()

Description

The **Next** method returns the next Portal object in the Portal collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A Portal object.

Example

```
&MyPortal = &MyCollection.Next();
```

Portal Collection Properties

In this section, we discuss the Count property.

Count

Description

This property returns the number of Portal objects in the Portal Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

NodeTemplate Class

The NodeTemplate object provides access to the default template assigned to a node in this portal.

A NodeTemplate object is instantiated by the First, InsertItem, ItemByName, and Next NodeTemplate Collection methods.

See Chapter 21, PortalRegistry Classes , NodeTemplate Collection, page 1148.

NodeTemplate Class Properties

In this section, we discuss the NodeTemplate class properties. The properties are discussed in alphabetical order.

DefaultTemplate

Description

This property specifies the name of the template to be applied to the node.

This property is read-write.

Name

Description

This property specifies the name of node that the template is to be applied to.

This property is read-write.

TemplateObject

Description

This property returns a reference to the template object associated with this node as a content reference.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Content Reference Class, page 1160.

NodeTemplate Collection

The NodeTemplate collection contains a set of references to each node template object in a portal.

A NodeTemplate collection is instantiated by the NodeTemplates PortalRegistry property.

See Chapter 21, PortalRegistry Classes , NodeTemplates, page 1133.

NodeTemplate Collection Methods

In this section, we discuss the NodeTemplate collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*NodeName*)

Description

The **DeleteItem** method deletes the NodeTemplate object identified by *NodeName* from the NodeTemplate Collection.

This method is not executed automatically. It is executed only when the PortalRegistry is saved.

Parameters

Parameter	Description
NodeName	Specify the name of a NodeTemplate to delete.

Returns

A Boolean value: True if the NodeTemplate was deleted, False otherwise.

Example

```
If Not &MyNodeTemplates.DeleteItem("HRMS") Then
   /* Do error processing */
End-if
```

First

Syntax

First()

Description

The **First** method returns the first NodeTemplate object in the NodeTemplate collection.

Parameters

None.

Returns

A NodeTemplate object.

Example

```
&MyNodeTemplate = &MyCollection.First();
```

InsertItem

Syntax

InsertItem(*NodeName*)

Description

The **InsertItem** method inserts the NodeTemplate object identified by *NodeName* into the NodeTemplate Collection.

This method is not executed automatically. It is executed only when the PortalRegistry is saved.

Parameters

Parameter	Description
NodeName	Specify the name of the node template to insert.

Returns

A reference to the new NodeTemplate object if the method executed successfully, NULL otherwise.

Example

```
&NewNodeTemplate = &MyPortal.NodeTemplates.InsertItem(“CRM”);
```

ItemByName

Syntax

ItemByName(*NodeName*)

Description

The **ItemByName** method returns the NodeTemplate object with the name *NodeName*.

Parameters

Parameter	Description
NodeName	Specify the name of an existing NodeTemplate object in the collection. If you specify an invalid name, the object is NULL.

Returns

A NodeTemplate object if successful, NULL otherwise.

Example

```
&MyNodeTemplate = &MyPortal.NodeTemplates.ItemByName("HRMS");
```

Next

Syntax

Next()

Description

The **Next** method returns the next NodeTemplate object in the NodeTemplate collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A NodeTemplate object.

Example

```
&MyNodeTemplate = &MyCollection.Next();
```

NodeTemplate Collection Properties

In this section, we discuss the Count property.

Count

Description

This property returns the number of NodeTemplate objects in the NodeTemplate Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Folder Class

Folder objects are instantiated from other classes as follows:

- From a PortalRegistry object with the RootFolder property or the FindFolderByName method.
- From a Folder Collection with the First, ItemByName, or Next methods

See Chapter 21, PortalRegistry Classes , RootFolder, page 1134; Chapter 21, PortalRegistry Classes , FindFolderByName, page 1123 and Chapter 21, PortalRegistry Classes , Folder Collection, page 1157.

See Chapter 21, PortalRegistry Classes , Adding a Folder, page 1248.

Folder Class Method

In this section, we discuss the Folder class methods. The methods are discussed in alphabetical order.

Save

Syntax

Save()

Description

The **Save** method saves any changes you made to the folder, for example, a changed description or ValidFrom date.

Using this method also saves any changes you made to PermissionValue or AttributeValue objects associated with this folder.

Parameters

None.

Returns

A Boolean value: True if the Folder object is successfully saved, False otherwise.

Example

```
If Not(&MyFolder.Save()) Then
   /* do error checking */
End-If;
```

See Also

Chapter 21, PortalRegistry Classes , Save, page 1131 PortalRegistry method
Chapter 21, PortalRegistry Classes , Save, page 1161 content reference method

Folder Class Properties

In this section, we discuss the Folder class properties. The properties are discussed in alphabetical order.

Attributes

Description

This property returns an Attribute Collection containing the AttributeValue objects for this folder.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Attribute Collection, page 1178.

Author

Description

This property returns the author (PeopleSoft user ID) for this folder as a string.

This property is read-only.

AuthorAccess

Description

This property specifies whether the author of the folder has access to the folder. This property takes a Boolean value. The default value for this property for a newly created object is True. This property is not cascaded.

This property is read-write.

Authorized

Description

This property specifies whether the user is authorized to view this Folder.

This property is used when you access a particular Folder using FindFolderByName. If you've specified a valid Folder with this method, a Folder is always returned, whether you are authorized to view it or not. This is the only property you can view from an object that you are not authorized to.

The initial value of this property depends on the other permission properties (PublicAccess and AuthorAccess) and the permission list values in the PermissionValue object associated with this folder.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , FindFolderByName, page 1123.

CascadedPermissions

Description

This property returns a PermissionValue Collection. This collection contains the value of the permissions for all the child and parent objects (up to the root folder). To determine only the permissions of the object use the Permission property instead.

Note. You *cannot* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can add values only to the collection returned by the Permissions property.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , PermissionValue Collection, page 1182, Chapter 21, PortalRegistry Classes , Permissions, page 1156.

ContentRefs

Description

This property returns the ContentReference Collection for this folder.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Content Reference Collection, page 1172.

CreationDate

Description

This property returns the creation date for this folder as a string.

This property is read-only.

Description

Description

This property returns or sets the description for this folder as a string.

The length of this property is 256 characters.

This property is read-write.

Folders

Description

This property returns a reference to the Folder Collection for this folder.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Folder Collection, page 1157.

IsMobile

Description

This property returns True if the folder is used with mobile applications, False otherwise.

This property is read-write.

IsVisible

Description

This property returns True if the Hide from Portal Navigation check box is *not* selected when the folder is created. If the folder is hidden from portal navigation, this property returns False.

Considerations Using IsVisible

If you do not specifically set this property on a new or copied folder, the IsVisible property is set as follows:

- If the ValidFrom date is less then or equal to that day’s date, the IsVisible property is set to True.
- If there is no ValidTo date the IsVisible property is set to True.
- If the ValidTo date is greater then or equal to that day’s date, the IsVisible property is set to True.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , ValidFrom, page 1157, Chapter 21, PortalRegistry Classes , ValidTo, page 1157.

Label

Description

This property returns or sets the label for this folder as a string.

The length of this property is 30 characters.

This property is translatable.

This property is read-write.

Name

Description

This property returns the name for this folder as a sting. The name is a unique identifier for each folder.

Every folder name must be unique across the portal, not just in the parent folder.

This property is *not* translatable. However, the value for the Label property is translatable.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Label, page 1155.

OwnerId

Description

This property returns the name of the owner for this folder as a string.

This property is read-write.

ParentName

Description

This property returns the parent folder name for this folder as a string. This property is valid only if the folder is contained within another folder. If the folder using this property is the root folder, this property returns -1.

This property is read-only.

Path

Description

The **Path** property returns a path to this folder, with each element of the path separated by a period. The path has the following syntax:

```
FolderLabel{Name}.[ChildFolderLabel{Name}.]. . .
```

This property is read-only.

Example

```
Departments{Root}.HR{EastCoast}.AdministerWorkforce{Global}
```

Permissions

Description

This property returns a PermissionValue Collection. This collection contains the value of the permissions for this folder. To determine the permissions for all the parent objects (up to the root folder) use the CascadedPermissions property.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , PermissionValue Collection, page 1182, Chapter 21, PortalRegistry Classes , CascadedPermissions, page 1153.

Product

Description

This property returns or sets the PeopleSoft product for this folder as a string.

The length of this property is 4 characters.

This property is read-write.

SequenceNumber

Description

The sequence number is used when returning a collection. The default order of the returned folders is based on the sequence number. Use this property to reorder folders.

If there are duplicates in the sequence number, the folders are returned alphabetically.

The length of this property is 4 characters.

This property is read-write.

PublicAccess

Description

This property indicates whether a folder is generally accessible, that is, if this property is set to True, any user can access the folder. This property is not cascaded.

This property takes a Boolean value.

The default value for this property for a newly created object is False.

This property is read-write.

ValidFrom

Description

This property returns or sets the date this folder is valid from as a string.

This property is read-write.

ValidTo

Description

This property returns or sets the date this folder is valid until as a string.

This property is read-write.

Note. The PortalRegistry API never uses the ValidTo and ValidFrom fields to determine what to return in a collection. You must check for these values in your application.

Folder Collection

The Folder Collection object provides access to a collection of folders in a Folder object.

The Folder Collection is instantiated from the Folders Folder Class property.

See Chapter 21, PortalRegistry Classes , Folders, page 1154.

Folder Collection Methods

In this section, we discuss the Folder collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

```
DeleteItem(FolderName)
```

Description

The **DeleteItem** method deletes the folder object identified by *FolderName* from the database. If the folder contains other folders, all child folders and their contents are also deleted.

Warning! If you delete a folder, you delete *all* content in the folder. If you delete a folder that contains other folders, that is, a parent folder, all the child folders, and all the content references are deleted.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for deletion, then actually deleted later. The item is deleted from the database *as soon as* the method is executed.

Parameters

Parameter	Description
FolderName	Specify the name of a folder existing in the folder collection.

Returns

A Boolean value: True if the folder was deleted, False otherwise.

Example

```
If Not &MyFolderColl.DeleteItem("MYFOLDER") Then
   /* Folder not deleted. Do error checking */
End-If;
```

First

Syntax

First()

Description

The **First** method returns the first Folder object in the folder collection.

Parameters

None.

Returns

Folder object.

Example

```
&MyFolder = &MyCollection.First();
```

InsertItem

Syntax

InsertItem(*FolderName*, *Label*)

Description

The **InsertItem** method inserts the folder object identified by *FolderName* from the Folder Collection. You must specify both a name and a label for all folders. This method returns a reference to the new folder object. You must specify a unique *FolderName*, or you receive an error.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for insertion, then actually inserted later. The item is inserted into the database *as soon as* the method is executed.

Parameters

Parameter	Description
FolderName	Specify the name of a folder existing in the folder collection. This parameter takes a string value.
Label	Specify a label for the new folder. This parameter takes a string value.

Returns

A reference to the new Folder object if the method executed successfully, False otherwise.

Example

```
&DeptHPFldr = &MyPortal.Folders.InsertItem("PORT0145", "HR Folder for ⇒
Department 0145");
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the Folder object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing folder within the folder collection. If you specify an invalid name, the object is NULL. You must specify a name, not a label.

Returns

A folder object if successful, NULL otherwise.

Example

```
&DeptFldr = &MyPortal.RootFolder.Folders.ItemByName(&Dept_Name);
```

Next

Syntax

Next()

Description

The **Next** method returns the next Folder object in the Folder collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A folder object.

Example

```
Local ApiObject &MySession, &Root, &Folders, &MyFolder;

&MySession = %Session;
&MyPortal = &MySession.GetPortalRegistry("ADMIN");

&Root = &MyPortal.GetRoot();
&Folders = &Root.Folders;

&MyFolder = &Folders.First();

For &I = 1 to &Folders.Count
   /* Do processing on folders */
   If &I <> &Folders.Count
      &MyFolder = &Folders.Next();
   End-If;
End-For;
```

Folder Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of Folder objects in the Folder Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Content Reference Class

The content reference object provides access to some kind of content. The type of content depends on the content reference.

The content reference objects are instantiated from other classes:

- From a PortalRegistry object with the FindCRefByURL or FindCRefByName properties.
- From a Content Reference Collection (instantiated from a folder) with the First, ItemByName or Next methods

See Chapter 21, PortalRegistry Classes , FindCRefByURL, page 1122; Chapter 21, PortalRegistry Classes , FindCRefByName, page 1121 and Chapter 21, PortalRegistry Classes , Content Reference Collection, page 1172.

See Chapter 21, PortalRegistry Classes , Adding a Content Reference, page 1250.

Content Reference Class Methods

In this section, we discuss the Content Reference class methods. The methods are discussed in alphabetical order.

Save

Syntax

Save()

Description

The **Save** method saves any changes you made to the content reference, for example, a changed description. It also performs some validation.

Using this method also saves any changes you made to PermissionValue or AttributeValue objects associated with this content reference.

Parameters

None.

Returns

A Boolean value: True if the content reference and its associated object saved successfully, False otherwise.

Example

```
If NOT(&MyCRef.Save()) Then
   /* save failed, do error processing */
End-If;
```

See Also

Chapter 21, PortalRegistry Classes , Save, page 1131 PortalRegistry method
Chapter 21, PortalRegistry Classes , Save, page 1151 Folder method

Content Reference Class Properties

In this section, we discuss the Content Reference class properties. The properties are discussed in alphabetical order.

AbsoluteContentURL

Description

This property returns the absolute content URL, that is, the content from the content servlet (psc).

This property is read-only.

Example

The following is an example absolute content URL:

```
http://serverx/servlets/psp/PS84/EMPLOYEEPORTAL/CRM/c/SFA.CUSTOMERINFO.GBL?page=⇒
CUST_DATA1&&Action=U&emplid=00001
```

AbsolutePortalURL

Description

This property returns the absolute content reference portal URL.

This property is read-only.

Example

```
http://serverx/servlets/psc/PS84/EMPLOYEEPORTAL/CRM/c/SFA.CUSTOMERINFO.GBL?page=⇒
CUST_DATA1&&Action=U&emplid=00001
```

AssignedPagelets

Description

This property returns an AssignedPagelet collection.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , AssignedPagelet Collection, page 1197.

Attributes

Description

This property returns an Attribute Collection containing the AttributeValues for this content reference object.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Attribute Collection, page 1178.

Author

Description

This property returns the author (PeopleSoft user ID) for this content reference object as a string.

This property is read-only.

AuthorAccess

Description

This property specifies whether the author of the content reference has access to the content reference. This property takes a Boolean value. The default value for this property is True.

This property is read-write.

Authorized

Description

This property specifies whether the user is authorized to view this content reference.

This property is used when you access a particular content reference using FindCRefByURL or FindCRefByName. If you specified a valid content reference with either of these methods, a content reference is always returned, whether you have authorized to view it or not. This is the only property you can view from an object that you are not authorized to.

The initial value of this property depends on the other permission properties (PublicAccess and AuthorAccess) and the permission list values in the PermissionValue object associated with this content reference.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , FindCRefByURL, page 1122, Chapter 21, PortalRegistry Classes , FindCRefByName, page 1121.

CascadedPermissions

Description

This property returns a PermissionValue Collection. This collection contains the value of the permissions for all the parent objects (up to the root folder). To determine only the permissions of the object use the Permissions property instead.

Note. You *cannot* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can add values only to the collection returned by the Permissions property.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , PermissionValue Collection, page 1182, Chapter 21, PortalRegistry Classes , Permissions, page 1166.

ContentProvider

Description

This property returns or sets the name of the node for the content reference as a string.

This property takes the following values:

Value	Description
Node name	Specify the exact name of the node.
LOCAL_NODE	Specify the node for the content reference as the node defined as the local node.

This property is read-write.

CreationDate

Description

This property returns the creation date for this content reference object as a string.

This property is read-only.

Data

Description

This property returns the data for this content reference. This property is valid only when the StorageType property is LOCL.

The length of this property depends on your system database limit for LONG fields.

This property is read-write.

Example

```
&MyData = &MyCRef.Data;
```

See Also

Chapter 21, PortalRegistry Classes , StorageType, page 1168.

Description

Description

This property returns or sets the description for this content reference object as a string.

The length of this property is 256 characters.

This property is read-write.

HtmlText

Description

This property returns the HTML text associated with this content reference as a string.

This property is read-only.

IsMobile

Description

This property returns True if this content reference is used with mobile applications, False otherwise.

This property is read-write.

IsVisible

Description

This property returns True if the Hide from Portal Navigation check box is *not* selected when the content reference is created. If the content reference is hidden from portal navigation, this property returns False.

Considerations Using IsVisible

If you do not specifically set this property on a new or copied folder, the IsVisible property is set as follows:

- If the ValidFrom date is less then or equal to that day's date, the IsVisible property is set to True.
- If there is no ValidTo date the IsVisible property is set to True.
- If the ValidTo date is greater then or equal to that day's date, the IsVisible property is set to True.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , ValidFrom, page 1172, Chapter 21, PortalRegistry Classes , ValidTo, page 1172.

Label

Description

This property returns or sets the label for this content reference object as a string.

The length of this property is 30 characters.

This property is translatable.

This property is read-write.

Name

Description

This property returns the name for this content reference object as a string. The name is a unique identifier for each content reference object.

Every content reference name must be unique in the portal, not just in the parent folder.

This property is read-only.

OwnerId

Description

This property returns or sets the owner ID of the content reference object as a string.

This property is read-write.

ParentName

Description

This property returns the parent folder name for this content reference object as a string.

This property is read-only.

Example

The following example uses the ParentName to return a folder object for the content reference.

```
&Folder = &Portal.FindFolderByName(&CRef.ParentName);
```

Path

Description

The **Path** property returns a path to this content reference, with each element of the path separated by a period. The path has the following syntax:

```
ContentReferenceLabel{Name}.[ChildContentReferenceLabel{Name}.]. . .
```

This property is read-only.

Permissions

Description

This property returns a PermissionValue Collection. This collection contains the value of the permissions for this content reference. To determine the permissions for all the parent objects (up to the root folder) use the CascadedPermissions property.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , PermissionValue Collection, page 1182, Chapter 21, PortalRegistry Classes , CascadedPermissions, page 1163.

Product

Description

This property returns or sets the PeopleSoft product for this content reference object as a string.

The length of this property is 4 characters.

This property is read-write.

PublicAccess

Description

This property indicates whether a folder is generally accessible, that is, if it will always be included in the general content reference collection. This property takes a Boolean value.

The default value for this property is False.

This property is read-write.

QualifiedURL

Description

Note. This property is being kept for backward compatibility only. If your code uses this property, the value returned is actually from the **AbsoluteContentURL** property. New applications should use the AbsoluteContentURL property instead.

See Also

Chapter 21, PortalRegistry Classes , AbsoluteContentURL, page 1162.

RelativeURL

Description

This property returns the relative URL in the following format:

```
../../../Portal/Node/Content_Type/ContentID
```

For example, from the following URL:

```
http://mlee2038/servlets/psp/PS84/e_procurement/fdm/c/E_PRO.CheckOut.GBL?page=⇒
view&Setid=110&Custid=99
```

The RelativeURL returns the following:

```
e_procurement/fdm/c/E_PRO.CheckOut.GBL?page=view&Setid=110&Custid=99
```

This property is read-only.

SequenceNumber

Description

The sequence number is used when returning a collection. The default order of the returned content references is based on the sequence number. Use this property to reorder content references. This property takes a number value.

If there are duplicates in the sequence number, the content references are returned alphabetically.

The length of this property is 4 characters.

This property is read-write.

StorageType

Description

In general, content references contain information about where to get the content, and do not store the content. However, content references that are template or portal component types can have their content accessible directly from the content reference. In these cases, the Data property is valid and can be read or written, and the data is stored locally in the portal database.

StorageType	Meaning
LOCL	Local: Data property on content reference is valid.
RMTE	Remote: Data property is not valid.

When UsageType is a target this property must be set to RMTE and the URLType property should specify what format the Node and URL are in.

When UsageType is either a template or a portal component this property can be set to either LOCL or RMTE.

Note that when StorageType is LOCL, it specifies a static template or portal component. But, when StorageType is RMTE it can specify either:

- a dynamically generated template or portal component
- a static template or portal component

In both cases, the Node and URL properties specifies how to get the template or portal component.

Only templates and homepage tab content references can have StorageType LOCL. The StorageType is always LOCL for homepage tabs. For templates LOCL means that corresponding URL is stored in database, not retreived by URL.

The following table indicates the usage type, and what type of storage is available.

UsageType	RMTE	LOCL
FRMT (Frame template)	X	X
HTMT (HTML template)	X	X
HPGT (Homepage tab)		X
HPGC (Pagemet)	X	
TARG (Target)	X	

RMTE is the default value for a new content reference.

The length of this property is 4 characters.

This property is read-write.

See Also

Chapter 21, PortalRegistry Classes , Data, page 1164.

Template

Description

This property returns or sets the name of the template used with this content reference as a string. You must specify the name of an existing template.

This property uses the name *not* the label of a content reference.

This property is used only when the UsageType property is specified as Target (TARG) and the TemplateType property is specified as HTML.

To return a reference to the content reference that contains the template specified by this property, use the TemplateObject property.

The length of this property is 30 characters.

This property is read-write.

See Also

Chapter 21, PortalRegistry Classes , UsageType, page 1171, Chapter 21, PortalRegistry Classes , TemplateType, page 1170, Chapter 21, PortalRegistry Classes , TemplateObject, page 1169.

TemplateObject

Description

This property returns a reference to a content reference object that contains the template specified by the Template property as a content reference. If no template is specified with Template, this property returns NULL.

This property is read-only.

See Also
Chapter 21, PortalRegistry Classes , Template, page 1169.

TemplateType

Description

This property indicates whether a template should be used to wrap the content. This property takes a string value.

Values are:

Value	Description
HTML	An HTML template should wrap the data (URL) in the content reference.
NONE	No template should be used. The text should not be wrapped.

HTML is the default value on a new content reference.

Use the NONE value if the URL should not appear in the portal. An example is when the content reference is a template itself.

If the homepage template for the user is un-retrievable, the system tries to use the default user's template first, before resorting to the portal default template.

When the content reference describes content, this property should be set to HTML.

This property is read-write.

See Also
Chapter 21, PortalRegistry Classes , Using Content References , page 1106.

URL

Description

This property returns or sets the URL for this content reference object as a string. The URL returns *exactly*as it appears in the database.

If you're setting a URL, you must use a unique URL.

The absolute URL, that is, the URL from the node concatenated with this URL must be unique.

You receive an error if the URL you use is already registered.

To retrieve a qualified URL, that is, one that contains the node URI, use the AbsoluteContentURL property.

The length of this property depends on your system database limit for LONG fields.

This property is read-write.

The format of the value for this property depends on the setting of other properties.

See Also

Chapter 21, PortalRegistry Classes , AbsoluteContentURL, page 1162, Chapter 21, PortalRegistry Classes , Using Content References , page 1106.

URLType

Description

This property indicates what kind of URL is used to retreive the content. This property takes a string value.

Note. This property is used only for content that has the StorageType property set as Remote.

Values are:

URLType Value	Meaning
UEXT	URL points to a non-PeopleSoft (external) URL
UMPG	URL points to a PeopleSoft mobile page
UPGE	URL points to a component.
UPHP	URL points to a homepage tab
UPTM	URL points to a template
USCR	URL points to an iScript
UGEN	URL points to a generic PeopleSoft URL

UPGE is the default value on a new content reference.

This property is read-write.

See Also

Chapter 21, PortalRegistry Classes , StorageType, page 1168.

UsageType

Description

This property indicates what the content reference is used for. Several other properties depend on this property.

This property takes a string value.

The length of this property is 4 characters.

Value	Description
FRMT	Frame template: the content reference is a frame-based template.
HTMT	HTML template: the content reference is an HTML template.
HPGT	Homepage tab: the content reference is a homepage tab.
HPGC	Pagelet: the content reference is a pagelet used in the homepage.
TARG	Target: the content reference is the target content its template determines what else must be loaded and how it will look.

TARG is the default value on a new content reference.

This property is read-write.

See Also

Chapter 21, PortalRegistry Classes , Using Content References , page 1106.

ValidFrom

Description

This property returns or sets the date this content reference is valid from as a string.

This property is read-write.

ValidTo

Description

This property returns or sets the date this content reference is valid until as a string.

This property is read-write.

Note. The PortalRegistry API never uses the ValidTo and ValidFrom fields to determine what to return in a collection. You must check for these values in your application.

Content Reference Collection

The Content Reference Collection provides access to the collection of content references in a Folder object.

The Content Reference Collection is instantiated from the ContentRefs Folder property.

See Chapter 21, PortalRegistry Classes , ContentRefs, page 1153.

Content Reference Collection Methods

In this section, we discuss the Content Reference collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*ContentReferenceName*)

Description

The **DeleteItem** method deletes the content reference object identified by *ContentReferenceName* from the content reference Collection.

If you delete a template for a content reference, and none of the other content references on a page have a template, the default template specified with the ContentProvider is used. If there's no template for the ContentProvider, the template for the PortalRegistry is used. However, if you delete the template for the content reference, the ContentProvider, and the PortalRegistry, you receive a runtime error.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for deletion, then actually deleted later. The item is deleted from the database *as soon as* the method is executed.

Parameters

Parameter	Description
ContentReferenceName	Specify the name of a content reference existing in the content reference collection.

Returns

A Boolean value: True if the content reference was deleted, False otherwise.

Example

```
If Not &MyCRef.DeleteItem("Test_CRef") Then
   /* can't delete test data. Do error processing */
End-if;
```

First

Syntax

First()

Description

The **First** method returns the first content reference object in the content reference collection.

Parameters

None.

Returns

A content reference object.

Example

```
&MyCRef = &MyCollection.First();
```

InsertItem

Syntax

InsertItem(*ContentReferenceName, ContentReferenceLabel, Node, URL*)

Description

The **InsertItem** method inserts the content reference object identified by *ContentReferenceName* into the content reference Collection.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for insertion, then actually inserted later. The item is inserted into the database as soon as the method is executed.

Parameters

Parameter	Description
ContentReferenceName	Specify the name of a new content reference. This parameter takes a string value. If you specify a name that already exists in the collection, you get an error.
ContentReferenceLabel	Specify a description of the new content reference. This is the translated value. This parameter takes a string value.
Node	Specify a node. This parameter takes a string value. If you specify a fully qualified value for *URL*, you can specify a NULL (that is, two quotation marks with no space between them (""")).
URL	Specify a URL that contains the content. The format of this parameter depends on other properties, such as the type of content reference, where the data is stored, and so on.

Returns

A reference to the new content reference object if the method executed successfully, NULL otherwise.

Example

The following example inserts an external content reference that is a template:

```
&URL = "t/" | &ITEMNAME;
&MyCRef = &CRefColl.InsertItem(&ITEMNAME, &ITEMLABEL, "", &URL);
```

The following example inserts a content reference where URLType is iScript (USCR):

```
&URL = "s/WEBLIB_PORTAL.FieldFormula.Portal_Trans_Dyn"

&MyCRef = &CRefColl.InsertItem(&ITEMNAME, &ITEMLABEL, "HRMS", &URL);
```

See Also

Chapter 21, PortalRegistry Classes , Using Content References , page 1106.

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the content reference object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing content reference within the content reference collection. If you specify an invalid name, the object is NULL.

Returns

A content reference object if successful, NULL otherwise.

Example

```
&MyCRef = &CRefColl.ItemByName("PORTAL_ADMIN);
```

Next

Syntax

Next()

Description

The **Next** method returns the next content reference object in the content reference collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

content reference object.

Example

```
&MyCRef = &MyCollection.Next();
```

Content Reference Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of content reference objects in the content reference Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

AttributeValue Class

The AttributeValue object provides access to attributes associated with either folders or content references.

AttributeValue objects are instantiated from an Attribute Collection Methods with the First, InsertItem, ItemByName, or Next methods.

See Chapter 21, PortalRegistry Classes , Attribute Collection Methods , page 1178.

See Chapter 21, PortalRegistry Classes , Using Attributes, page 1255.

AttributeValue Class Properties

In this section, we discuss the AttributeValue class properties. The properties are discussed in alphabetical order.

Label

Description

This property returns the label of the AttributeValue as a string. This property works with the **Translatable** property. If Translatable is set to True, the value of Label can be translated.

The length of this property is 30 characters.

This property is read-write.

See Also

Chapter 21, PortalRegistry Classes , Translatable, page 1177.

Name

Description

This property returns the name of the AttributeValue as a string.

The length of this property is 30 characters.

This property is read-only.

Example

```
&AttrColl = &Folder.Attributes;
&Attr = &AttrColl.First();

   &Scroll = GetLevel0().GetRow(1).GetRowset(Scroll.PORTAL_FLDR_ATR);

   &I = 1;
   While All(&Attr)
      &Record = &Scroll.GetRow(&I).GetRecord(Record.PORTAL_FLDR_ATR);
      &Record.PORTAL_ATTR_NAM.Value = &Attr.Name;
      &Record.PORTAL_ATTR_VAL.Value = &Attr.Value;
      &Attr = &AttrColl.Next();
      /* need this check so we don't insert extra blank row */
      If All(&Attr) Then
         &Scroll.InsertRow(&I);
         &I = &I + 1;
      End-If;
End-While;
```

Translatable

Description

This property specifies if the AttributeValue is translatable. This property takes a Boolean value: True if the AttributeValue can be translated, False otherwise.

If this property is set to True, the value of the Label property can be translated.

Note. Regardless of the order in which attributes were entered, they are ordered according to their translatable property, that is, attributes that have this property set as True come first, followed by attributes that have this property set as False.

This property is read-write.

See Also

Chapter 21, PortalRegistry Classes , Label, page 1176.

Value

Description

This property returns the value of the AttributeValue as a string.

The length of this property depends on your system database limit for LONG fields.

This property is read-write.

Example

To specify more than a single value for an AttributeValue, you can specify several values separated by a semicolon. For example:

```
&MyAtt.value = "401k;benefits;dependants;HR";
```

Attribute Collection

The Attribute Collection object provides access to the collection of Attribute in a Folder or a content reference object.

An Attribute Collection is instantiated from other classes as follows:

- From a content reference object with the Attributes property.
- From a Folder object with the Attributes property.
- From a PageletCategory object with the Attributes property.
- From a Pagelet object with the Attributes property.

See Chapter 21, PortalRegistry Classes , Attributes, page 1162.

See Chapter 21, PortalRegistry Classes , Attributes, page 1152.

See Chapter 21, PortalRegistry Classes , Attributes, page 1211.

See Chapter 21, PortalRegistry Classes , Attributes, page 1218.

Attribute Collection Methods

In this section, we discuss the Attribute collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*AttributeValueName*)

Description

The **DeleteItem** method deletes the AttributeValue object identified by *AttributeValueName* from the Attribute Collection.

This method is not executed automatically. It is executed only when the parent object is saved.

Parameters

Parameter	Description
AttributeValueName	Specify the name of an AttributeValue existing in the Attribute Collection.

Returns

A Boolean value: True if the AttributeValue was deleted, False otherwise.

First

Syntax

```
First()
```

Description

The **First** method returns the first AttributeValue object in the Attribute Collection.

Parameters

None.

Returns

An AttributeValue object.

Example

```
&MyAttributeValue = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(AttributeValueName)
```

Description

The **InsertItem** method inserts the AttributeValue object identified by *AttributeValueName* from the Attribute Collection.

This method is not executed automatically. It is executed only when the parent object is saved.

Parameters

Parameter	Description
AttributeValueName	Specify the name of an AttributeValue existing in the Attribute Collection.

Returns

A reference to the new AttributeValue object if the method executed successfully, NULL otherwise.

Example

```
For &I = 1 To &Rowset.ActiveRowCount
    If &CompName = "PORTAL_CREF_ADM" Then
       &Record = &Rowset.GetRow(&I).GetRecord(Record.PORTAL_CATR_DV);
    Else
       &Record = &Rowset.GetRow(&I).GetRecord(Record.PORTAL_FATR_DV);
    End-If;

    If &Record.PORTAL_ATTR_NAM.Value <> "" Then
       &Attr = &AttrColl.InsertItem(&Record.PORTAL_ATTR_NAM.Value);
       &Attr.Value = &Record.PORTAL_ATTR_VAL.Value;
    End-If;
End-For;
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the AttributeValue object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing AttributeValue within the Attribute Collection. If you specify an invalid name, the object is NULL.

Returns

An AttributeValue object if successful, NULL otherwise.

Example

```
&Attr = &AttrColl.ItembyName("RELLINK");
```

Next

Syntax

Next()

Description

The **Next** method returns the next AttributeValue object in the Attribute Collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

AttributeValue object.

Example

```
&MyAttributeValue = &MyCollection.Next();
```

Attribute Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of AttributeValue objects in the Attribute Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

PermissionValue Class

The PermissionValue object provides access to permission lists associated with folders, content references, PageletCategory objects, or Pagelets.

Note. You *cannot* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can add values only to the collection returned by the Permissions property.

PermissionValue objects are instantiated from a PermissionValue Collection with the First, InsertItem, ItemByName, or Next methods.

See Chapter 21, PortalRegistry Classes , PermissionValue Collection, page 1182.

See Chapter 21, PortalRegistry Classes , Using Security , page 1097.

See Chapter 21, PortalRegistry Classes , Setting Permissions Using the PermissionValue Object, page 1253.

PermissionValue Class Properties

In this section we discuss the Cascade and Name properties.

Cascade

Description

This property indicates whether the current permission should be granted to all child folders.

This property is valid only with folders, not with content references.

This property takes a Boolean value. The default value for a new PermissionValue object is False.

This property is read-write.

Name

Description

Specify the name of a permission list as the name of this object. You must specify a permission list that has already been created. This property takes a string value.

The length of this property is 30 characters.

This property is read-write.

PermissionValue Collection

A PermissionValue collection is returned by the following:

- CascadedPermissions and Permissions folder property
- CascadedPermissions and Permissions content reference property
- CascadedPermissions and Permissions PageletCategory object property
- CascadedPermissions and Permissions Pagelet property

What is contained in the PermissionValue collection depends on the property that created it.

See Chapter 21, PortalRegistry Classes , CascadedPermissions, page 1153 and Chapter 21, PortalRegistry Classes , Permissions, page 1156.

See Chapter 21, PortalRegistry Classes , CascadedPermissions, page 1163 and Chapter 21, PortalRegistry Classes , Permissions, page 1166.

See Chapter 21, PortalRegistry Classes , CascadedPermissions, page 1212 and Chapter 21, PortalRegistry Classes , Permissions, page 1213.

See Chapter 21, PortalRegistry Classes , CascadedPermissions, page 1219 and Chapter 21, PortalRegistry Classes , Permissions, page 1222.

See Chapter 21, PortalRegistry Classes , Using Security , page 1097.

PermissionValue Collection Methods

In this section, we discuss the PermissionValue collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*PermissionValueName*)

Description

The **DeleteItem** method deletes the PermissionValue object identified by *PermissionValueName* from the PermissionValue Collection.

This method is not executed automatically. It is executed only when the parent object is saved.

Parameters

Parameter	Description
PermissionValueName	Specify the name of a PermissionValue existing in the PermissionValue collection.

Returns

A Boolean value: True if the PermissionValue was deleted, False otherwise.

Example

```
If Not &MyPValColl.DeleteItem("ALLPNLS") Then
   /* do error processing */
End-If;
```

First

Syntax

First()

Description

The **First** method returns the first PermissionValue object in the PermissionValue collection.

Parameters

None.

Returns

A PermissionValue object.

Example

```
&MyPermissionValue = &MyCollection.First();
```

InsertItem

Syntax

InsertItem(*PermissionValueName*)

Description

The **InsertItem** method inserts the PermissionValue object identified by *PermissionValueName* into the PermissionValue Collection.

This method is not executed automatically. It is executed only when the parent object is saved.

Note. You *cannot* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can only add values to the collection returned by the Permissions property.

Parameters

Parameter	Description
PermissionValueName	Specify the name of an existing permission list.

Returns

A reference to the new PermissionValue object if the method executed successfully, NULL otherwise.

Example

```
&MyPermV = &MyPermVColl.InsertItem("ALLPNLS");

If Not &MyPermV Then
   /* do error processing */
End-If;
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the PermissionValue object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing PermissionValue within the PermissionValue collection. If you specify an invalid name, the object is NULL.

Returns

A PermissionValue object if successful, NULL otherwise.

Example

```
&MyPVal = &MyPValColl.ItemByName("CUSTOMER");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next PermissionValue object in the PermissionValue collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

PermissionValue object.

Example

```
&MyPermissionValue = &MyCollection.Next();
```

PermissionValue Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of PermissionValue objects in the PermissionValue Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

TabDefinition Class

The TabDefinition object provides access to the homepage tab defined for the portal.

TabDefinition objects are instantiated from a TabDefinition Collection with the First, InsertItem, ItemByName, or Next methods.

See Chapter 21, PortalRegistry Classes , TabDefinition Collection, page 1192.

TabDefinition Class Methods

In this section, we discuss the Save method.

Save

Syntax

Save()

Description

The **Save** method saves any changes you made to the TabDefinition, for example, a changed description. It also performs some validation.

Using this method also saves any changes you made to DynamicCategories and AssignedPagelets objects associated with this TabDefinition.

Parameters

None.

Returns

A Boolean value: True if the TabDefinition and its associated object saved successfully, False otherwise.

Example

```
If NOT(&MyTabDefn.Save()) Then
   /* save failed, do error processing */
End-If;
```

See Also

Chapter 21, PortalRegistry Classes , Save, page 1131 PortalRegistry method
Chapter 21, PortalRegistry Classes , Save, page 1151 Folder method
Chapter 21, PortalRegistry Classes , Save, page 1161 ContentReference method

TabDefinition Class Properties

In this section, we discuss the TabDefinition class properties. The properties are discussed in alphabetical order.

AssignedPagelets

Description

This property returns a reference to an AssignedPagelet Collection for this TabDefinition object.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , AssignedPagelet Collection, page 1197.

AvailableCategories

Description

This property returns a reference to an AvailableCategory Collection for this TabDefinition object.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , AvailableCategory Collection, page 1201.

AvailablePagelets

Description

This property returns a reference to an AvailablePagelet Collection for this TabDefinition object.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , AvailablePagelet Collection, page 1205.

Attributes

Description

This property returns an Attribute Collection containing the AttributeValues for this TabDefinition object.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Attribute Collection, page 1178.

Author

Description

This property returns the author (PeopleSoft user ID) for this TabDefinition object as a string.

This property is read-only.

AuthorAccess

Description

This property specifies whether the author of the TabDefinition has access to the TabDefinition. This property takes a Boolean value. The default value for this property is True.

This property is read-write.

ColumnLayout

Description

This property indicates whether the tab layout is for two or three columns. This property takes a string value.

The values for this property are:

Value	Description
2	Two-column table
3	Three-column table

This property is read-write.

CreationDate

Description

This property returns the creation date for this TabDefinition object as a string.

This property is read-only.

Description

Description

This property returns or sets the description for this TabDefinition object as a string.

The length of this property is 256 characters.

This property is read-write.

DynamicCategories

Description

This property returns a reference to a DynamicCategory Collection for this TabDefinition object.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , DynamicCategory Collection, page 1207.

HelpID

Description

This property returns the help system identifier for this tab. It enables the help system to provide tab-specific help. This property takes a string value.

This property is read-write.

HtmlText

Description

This property returns the HTML text associated with this tab definition as a string.

This property is read-only.

IsHideActionBar

Description

This property hides all pagelet action bar images for all pagelets used on this tab. This property overrides any properties for the pagelets used on this tab. This property takes a Boolean value: True, hide images, False, display images

This property is read-write.

IsLayoutLocked

Description

This property indicates whether the user can change the tab's default number of columns. This property takes a Boolean value: True, the user can change the tab columns, False, the columns can't be changed by the user.

This property is read-write.

IsRenameable

Description

This property indicates whether the tab's label can be changed by the user. This property takes a Boolean value: True, the user can change the label, False, the user can't.

This property is read-write.

Label

Description

This property returns or sets the label for this TabDefinition object as a string.

The length of this property is 30 characters.

This property is translatable.

This property is read-write.

LayoutBehavior

Description

This property indicates whether the user can change when and how the tab displays. This property takes a string value.

The values are:

Value	Description
4OPT	The user can choose to add this tab to their homepage. It doesn't show up by default.
3DEF	The user sees this tab the first time they log in, however, they can remove this tab if they'd like.
2REQ	The user can't remove this tab from their homepage. However, they are allowed to change its sequence or order on the page.
1FIX	The user can't remove this tab or change its position on the page.

This property is read-write.

Name

Description

This property returns the name for this TabDefinition object as a string. The name is a unique identifier for each TabDefinition object.

Every TabDefinition name must be unique in the portal, not just in the parent folder.

This property is read-only.

OwnerId

Description

This property returns or sets the owner ID of the TabDefinition object as a string.

This property is read-write.

Product

Description

This property returns or sets the PeopleSoft product for this TabDefinition object as a string.

The length of this property is 4 characters.

This property is read-write.

PublicAccess

Description

This property indicates whether a TabDefinition is generally accessible, that is, if it is always included in the general TabDefinition collection. This property takes a Boolean value.

The default value for this property is False.

This property is read-write.

QualifiedURL

Description

This property returns an absolute URL. If the TabDefinition has a ContentProvider associated with it, the URI from the ContentProvider is concatenated with the URL from the TabDefinition. If there is no ContentProvider, this property returns the text in the URL property.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , URL, page 1170.

SequenceNumber

Description

The sequence number is used when returning a collection. The default order of the returned TabDefinitions is based on the sequence number. Use this property to reorder TabDefinitions. This property takes a number value.

If there are duplicates in the sequence number, the TabDefinitions are returned alphabetically.

The length of this property is 4 characters.

This property is read-write.

StyleSheet

Description

This property defines the style sheet to use for this tab. This property takes a string value. This property is 30 characters long.

If no style sheet is specified, the default style sheet PSSTYLEDEF is used.

This property is read-write.

ValidFrom

Description

This property returns or sets the date this TabDefinition is valid from as a string.

This property is read-write.

ValidTo

Description

This property returns or sets the date this TabDefinition is valid until as a string.

This property is read-write.

Note. The PortalRegistry API never uses the ValidTo and ValidFrom fields to determine what to return in a collection. You must check for these values in your application.

TabDefinition Collection

A TabDefinition collection is the collection of all TabDefinitions for a portal.

A TabDefinition collection is returned by the TabDefinitions PortalRegistry property.

See Chapter 21, PortalRegistry Classes , TabDefinitions, page 1134.

TabDefinition Collection Methods

This section describes the TabDefinition collection methods in alphabetical order.

DeleteItem

Syntax

```
DeleteItem(TabDefinitionName)
```

Description

The **DeleteItem** method deletes the TabDefinition object identified by *TabDefinitionName* from the TabDefinition Collection.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for deletion, then actually deleted later. The item is deleted from the database *as soon as* the method is executed.

Parameters

Parameter	Description
TabDefinitionName	Specify the name of a TabDefinition existing in the TabDefinition collection.

Returns

A Boolean value: True if the TabDefinition was deleted, False otherwise.

Example

```
If Not &MyColl.DeleteItem("TabDefnTest") Then
   /* do error processing */
End-If;
```

First

Syntax

First()

Description

The **First** method returns the first TabDefinition object in the TabDefinition collection.

Parameters

None.

Returns

A TabDefinition object.

Example

```
&MyTabDefinition = &MyCollection.First();
```

InsertItem

Syntax

InsertItem(*TabDefinitionName*)

Description

The **InsertItem** method inserts the TabDefinition object identified by *TabDefinitionName* into the TabDefinition Collection.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for insertion, then actually inserted later. The item is inserted into the database *as soon as* the method is executed.

Parameters

Parameter	Description
TabDefinitionName	Specify the name of an existing permission list.

Returns

A reference to the new TabDefinition object if the method executed successfully, NULL otherwise.

Example

```
&MyTabDef = &MyColl.InsertItem("Empl_Homepage");

If Not &MyTabDef Then
   /* do error processing */
End-If;
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the TabDefinition object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing TabDefinition within the TabDefinition collection. If you specify an invalid name, the object is NULL.

Returns

A TabDefinition object if successful, NULL otherwise.

Example

```
&MyTebDefn = &MyColl.ItemByName("Empl_Homepage");
```

Next

Syntax

Next()

Description

The **Next** method returns the next TabDefinition object in the TabDefinition collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

TabDefinition object.

Example

```
&MyTabDefinition = &MyCollection.Next();
```

TabDefinition Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of TabDefinition objects in the TabDefinition Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

AssignedPagelet Class

An AssignedPagelet is a pagelet object that has been assigned to a particular tab. This pagelet also contains the layout data that is specific to the pagelet and tab.

AssignedPagelet objects are instantiated from an AssignedPagelet Collection with the First, InsertItem, ItemByName, or Next methods.

See Chapter 21, PortalRegistry Classes , AssignedPagelet Collection, page 1197.

AssignedPagelet Class Properties

In this section, we discuss the AssignedPagelet class properties. The properties are discussed in alphabetical order.

Column

Description

This property returns the column in which this pagelet is displayed, as a number.

This property is read-write.

LayoutBehavior

Description

This property indicates whether the user can change when and how the pagelet displays. This property takes a string value.

The values are:

Value	Description
4OPT	The user can choose to add this pagelet to their tab. It doesn't show up by default. The user can remove and reposition this pagelet on the tab.
3DEF	The user sees this pagelet the first time they log in, however, they can remove or reposition this pagelet if they'd like.
2REQ	The user can't remove this pagelet from the tab. However, they are allowed to reposition the pagelet on the tab.
1FIX	The user can't remove this pagelet or change its position on the tab.

This property is read-write.

PageletName

Description

This property returns the name for this Pagelet object as a string. The name is a unique identifier for each Pagelet object.

This property is read-only.

Row

Description

This property returns the row in which this pagelet is displayed, as a number.

This property is read-write.

AssignedPagelet Collection

An AssignedPagelet collection is returned by the AssignedPagelet TabDefinition method.

It contains a collection of all the pagelets explicitly assigned to the tab.

See Chapter 21, PortalRegistry Classes , AssignedPagelets, page 1187.

See Chapter 21, PortalRegistry Classes , AssignedPagelet Class, page 1195.

AssignedPagelet Collection Methods

In this section, we discuss the AssignedPagelet collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*PageletName*)

Description

The **DeleteItem** method deletes the AssignedPagelet object identified by *PageletName* from the AssignedPagelet Collection.

This method is not executed automatically. It is executed only when the parent object is saved.

Parameters

Parameter	Description
PageletName	Specify the name of an AssignedPagelet existing in the AssignedPagelet collection.

Returns

A Boolean value: True if the AssignedPagelet was deleted, False otherwise.

Example

```
If Not &MyColl.DeleteItem("Weather_411") Then
   /* do error processing */
End-If;
```

First

Syntax

First()

Description

The **First** method returns the first AssignedPagelet object in the AssignedPagelet collection.

Parameters

None.

Returns

An AssignedPagelet object.

Example

```
&MyAssignedPagelet = &MyCollection.First();
```

InsertItem

Syntax

InsertItem(*PageletName*, *Column*, *Row*, *LayoutBehavior*)

Description

The **InsertItem** method inserts the AssignedPagelet object identified by the parameters into the AssignedPagelet Collection.

This method is not executed automatically. It is executed only when the parent object is saved.

Parameters

Parameter	Description
PageletName	Specify the name of the pagelet that you want to insert.
Column	Specify the column that you want this pagelet to display in, as a number.
Row	Specify the row that you want this pagelet to display in, as a number. You can specify a zero if you don't know which row.
LayoutBehavior	Specify whether the user can change when and how the pagelet displays. This property takes a string value. The values are:

Value	Description
4OPT	The user can choose to add this pagelet to their tab. It doesn't show up by default. The user can remove and reposition this pagelet on the tab.
3DEF	The user sees this pagelet the first time they log in, however, they can remove or reposition this pagelet if they'd like.
2REQ	The user can't remove this pagelet from the tab. However, they are allowed to reposition the pagelet on the tab.
1FIX	The user can't remove this pagelet or change its position on the tab.

Returns

A reference to the new AssignedPagelet object if the method executed successfully, NULL otherwise.

Example

```
&MyPagelet = &MyColl.InsertItem("Weather_411");

If Not &MyPagelet Then
   /* do error processing */
End-If;
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the AssignedPagelet object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing AssignedPagelet within the AssignedPagelet collection. If you specify an invalid name, the object is NULL.

Returns

An AssignedPagelet object if successful, NULL otherwise.

Example

```
&MyPagelet = &MyColl.ItemByName("Dictionary");
```

Next

Syntax

Next()

Description

The **Next** method returns the next AssignedPagelet object in the AssignedPagelet collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

An AssignedPagelet object.

Example

```
&MyAssignedPagelet = &MyCollection.Next();
```

AssignedPagelet Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of AssignedPagelet objects in the AssignedPagelet Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

AvailableCategory Class

An AvailableCategory class is composed of a category name and an AvailablePagelet Collection, that is, a collection of all the available pagelets associated with that category.

AvailableCategory objects are instantiated from an AvailableCategory Collection with the First, ItemByName, or Next methods.

See Chapter 21, PortalRegistry Classes , AvailableCategory Collection, page 1201.

AvailableCategory Class Properties

In this section, we discuss the AvailableCategory class properties. The properties are discussed in alphabetical order.

AvailablePagelets

Description

This property returns a reference to an AvailablePagelet Collection associated with the specified CategoryName.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , AvailablePagelet Collection, page 1205.

CategoryName

Description

Use this property to return the name of a category to be associated with the TabDefinition, as a string.

This property is read-only.

AvailableCategory Collection

An AvailableCategory is composed of a category name and an AvailablePagelet Collection, that is, a collection of all the available pagelets associated with that category.

An AvailableCategory collection is returned by the AvailableCategories TabDefinition property.

See Chapter 21, PortalRegistry Classes , AvailableCategory Collection, page 1201.

AvailableCategory Collection Methods

In this section, we discuss the AvailableCategory collection methods. The methods are discussed in alphabetical order.

First

Syntax

First()

Description

The **First** method returns the first AvailableCategory object in the AvailableCategory collection.

Parameters

None.

Returns

An AvailableCategory object.

Example

```
&MyAvailableCategory = &MyCollection.First();
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the AvailableCategory object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing AvailableCategory within the AvailableCategory collection. If you specify an invalid name, the object is NULL.

Returns

An AvailableCategory object if successful, NULL otherwise.

Example

```
&MyCat = &MyColl.ItemByName("Dictionary");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next AvailableCategory object in the AvailableCategory collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

AvailableCategory object.

Example

```
&MyAvailableCategory = &MyCollection.Next();
```

AvailableCategory Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of AvailableCategory objects in the AvailableCategory Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

AvailablePagelet Class

An AvailablePagelet is a pagelet object can be assigned to a particular tab. This includes all pagelets from the dynamic categories, and assigned pagelets. Each pagelet also contains the layout data that is specific to that pagelet.

AvailablePagelet objects are instantiated from an AvailablePagelet Collection with the First, ItemByName, or Next methods.

See Chapter 21, PortalRegistry Classes , AvailablePagelet Collection, page 1205.

AvailablePagelet Class Properties

In this section, we discuss the AvailablePagelet class properties. The properties are discussed in alphabetical order.

CategoryLabel

Description

This property turns the label of the category to which the pagelet is assigned, as a string.

This property is read-only.

CategoryName

Description

This property returns the name of the category to which the pagelet is assigned, as a string.

This property is read-only.

Column

Description

This property returns the column in which this pagelet is displayed, as a number.

This property is read-only.

LayoutBehavior

Description

This property indicates whether the user can change when and how the pagelet displays. This property takes a string value.

The values are:

Value	Description
4OPT	The user can choose to add this pagelet to their tab. It doesn't show up by default. The user can remove and reposition this pagelet on the tab.
3DEF	The user sees this pagelet the first time they log in, however, they can remove or reposition this pagelet if they'd like.
2REQ	The user can't remove this pagelet from the tab. However, they are allowed to reposition the pagelet on the tab.
1FIX	The user can't remove this pagelet or change its position on the tab.

This property is read only.

PageletLabel

Description

This property returns the label for this property as a string.

This property is read-only.

PageletName

Description

This property returns the name for this Pagelet object as a string. The name is a unique identifier for each Pagelet object.

This property is read-only.

Row

Description

This property returns the row in which this pagelet is displayed, as a number.

This property is read-only.

AvailablePagelet Collection

An AvailablePagelet collection is returned by the AvailablePagelet TabDefinition method.

It contains a collection of all the pagelets that could be assigned to the tab. It's an aggregation of the pagelets in the AssignedPagelets collection and all the pagelets in each of the categories in the DynamicCategories collection. Pagelet category sequence numbers, followed by pagelet sequence numbers, then name, sort this collection.

If you have two pagelets assigned with the same name (that is, an assigned pagelet and a dynamic category pagelet), the assigned pagelet takes precedence over the dynamic category one, and is the only one listed.

See Chapter 21, PortalRegistry Classes , AvailablePagelets, page 1187.

See Chapter 21, PortalRegistry Classes , AvailablePagelet Class, page 1203.

AvailablePagelet Collection Methods

In this section, we discuss the AvailablePagelet collection methods. The methods are discussed in alphabetical order.

First

Syntax

First()

Description

The **First** method returns the first AvailablePagelet object in the AvailablePagelet collection.

Parameters

None.

Returns

An AvailablePagelet object.

Example

```
&MyAvailablePagelet = &MyCollection.First();
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the AvailablePagelet object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing AvailablePagelet within the AvailablePagelet collection. If you specify an invalid name, the object is NULL.

Returns

An AvailablePagelet object if successful, NULL otherwise.

Example

```
&MyPagelet = &MyColl.ItemByName("Dictionary");
```

Next

Syntax

Next()

Description

The **Next** method returns the next AvailablePagelet object in the AvailablePagelet collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

AvailablePagelet object.

Example

```
&MyAvailablePagelet = &MyCollection.Next();
```

AvailablePagelet Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of AvailablePagelet objects in the AvailablePagelet Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

DynamicCategory Collection

A DynamicCategory collection contains a collection of PageletCategory names associated with a TabDefinition. The collection is initially ordered by category name.

When a PageletCategory is dynamic, it is immediately available to an end-user.

A DynamicCategory Collection is instantiated by the DynamicCategories property of a TabDefinition.

See Chapter 21, PortalRegistry Classes , DynamicCategories, page 1189.

DynamicCategory Collection Methods

In this section, we discuss the DynamicCategory collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*Name*)

Description

The **DeleteItem** method deletes the PageletCategory object identified by *Name* from the DynamicCategory Collection. This does *not* delete the PageletCategory from the database, just from the DynamicCategory collection.

This method is not executed automatically. It is executed only when the parent object is saved.

Note. If you delete a DynamicCategory, tabs with the attribute PORTAL_HPTAB_DYN lists must be searched for and updated (if necessary) to reflect the deleted category.

Parameters

Parameter	Description
Name	Specify the name of an existing category in the DynamicCategory collection.

Returns

A Boolean value: True if the PageletCategory was deleted from the DynamicCategory collection, False otherwise.

Example

```
If Not &MyColl.DeleteItem("Dictionaries") Then
   /* do error processing */
End-If;
```

First

Syntax

First()

Description

The **First** method returns the name of the first PageletCategory in the DynamicCategory collection.

Parameters

None.

Returns

A string.

Example

```
&MyDynamicCategory = &MyCollection.First();
```

InsertItem

Syntax

InsertItem(*Name*)

Description

The **InsertItem** method inserts the PageletCategory object identified by *Name* into the DynamicCategory Collection. This does *not* insert the PageletCategory into the database, just into the DynamicCategory collection. After a PageletCategory is marked as dynamic, it is immediately available to the end-user.

This method is not executed automatically. It is executed only when the parent object is saved.

Parameters

Parameter	Description
Name	Specify the name of the DynamicCategory to insert.

Returns

A string containing the name of the new DynamicCategory object if the method executed successfully, Null otherwise.

Example

```
&MyCat = &MyColl.InsertItem("User_Info");
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the name of the PageletCategory object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing DynamicCategory within the DynamicCategory collection. If you specify an invalid name, this method returns a Null string.

Returns

A string.

Example

```
&MyPagelet = &MyColl.ItemByName("Dictionary");
```

Next

Syntax

Next()

Description

The **Next** method returns the name of the next PageletCategory object in the DynamicCategory collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A string.

Example

```
&MyDynamicCategory = &MyCollection.Next();
```

DynamicCategory Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of PageletCategory names in the DynamicCategory Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

PageletCategory Class

A PageletCategory is a group of related pagelets, such as Weather, News, Reference, and so on. Each PageletCategory has a collection of pagelets associated with it.

A PageletCategory is instantiated from a PageletCategory Collection with the First, InsertItem, ItemByName, or Next methods.

All PageletCategory objects that are dynamic are contained in the DynamicCategory collection. The methods from this collection return the names of the PageletCategory objects only, not references to the objects themselves.

See Chapter 21, PortalRegistry Classes , PageletCategory Collection, page 1214.

PageletCategory Class Method

In this section, we discuss the Save method.

Save

Syntax

Save()

Description

The **Save** method saves any changes you made to the PageletCategory, for example, a changed description or sequence number.

Parameters

None.

Returns

A Boolean value: True if the PageletCategory object is successfully saved, False otherwise.

Example

```
If Not(&MyPageletCategory.Save()) Then
   /* do error checking */
End-If;
```

See Also

Chapter 21, PortalRegistry Classes , Save, page 1131 PortalRegistry method
Chapter 21, PortalRegistry Classes , Save, page 1161 Folder method

PageletCategory Class Properties

In this section, we discuss the PageletCategory class properties. The properties are discussed in alphabetical order.

Attributes

Description

This property returns an Attribute Collection containing the AttributeValues for this PageletCategory object.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Attribute Collection, page 1178.

Author

Description

This property returns the author (PeopleSoft user ID) for this PageletCategory as a string.

This property is read-only.

AuthorAccess

Description

This property specifies whether the author of the PageletCategory has access to the PageletCategory. This property takes a Boolean value. The default value for this property for a newly created object is True. This property is not cascaded.

This property is read-write.

Authorized

Description

This property specifies whether the user is authorized to view this PageletCategory.

The initial value of this property depends on the other permission properties (PublicAccess and AuthorAccess) and the permission list values in the PermissionValue object associated with this PageletCategory.

This property is read-only.

CascadedPermissions

Description

This property returns a PermissionValue Collection. This collection contains the value of the permissions for all the child and parent objects (up to the root PageletCategory). To determine only the permissions of the object use the Permission property instead.

Note. The PORTAL_PAGELETS folder is the parent folder for all PageletCategories. Its security is set to public. PeopleSoft does not recommend cascading any permissions from this folder object. Cascade permissions only from the pagelet category (folder). You *cannot* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can add values only to the collection returned by the Permissions property.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , PermissionValue Collection, page 1182, Chapter 21, PortalRegistry Classes , Permissions, page 1156.

CreationDate

Description

This property returns the creation date for this PageletCategory as a string.

This property is read-only.

Description

Description

This property returns or sets the description for this PageletCategory as a string.

The length of this property is 256 characters.

This property is read-write.

Label

Description

This property returns or sets the label for this PageletCategory as a string.

The length of this property is 30 characters.

This property is translatable.

This property is read-write.

Name

Description

This property returns the name for this PageletCategory as a sting. The name is a unique identifier for each PageletCategory.

Every PageletCategory name must be unique in across the portal, not just in the parent PageletCategory.

This property is *not* translatable. However, the value for the Label property is translatable.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Label, page 1212.

OwnerId

Description

This property returns or sets the owner ID of the PageletCategory object as a string.

This property is read-write.

Pagelets

Description

This property returns a reference to a Pagelet Collection for this PageletCategory.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Pagelet Collection, page 1224.

Permissions

Description

This property returns a PermissionValue Collection. This collection contains the value of the permissions for this PageletCategory.

Note. The PORTAL_PAGELETS folder is the parent folder for all PageletCategories. Its security is set to public. PeopleSoft does not recommend cascading any permissions from this folder object. Cascade permissions only from the pagelet category (folder).

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , PermissionValue Collection, page 1182.

Product

Description

This property returns or sets the PeopleSoft product for this PageletCategory as a string.

The length of this property is 4 characters.

This property is read-write.

PublicAccess

Description

This property indicates whether a PageletCategory is generally accessible, that is, if this property is set to True, any user can access the PageletCategory. This property is not cascaded.

This property takes a Boolean value.

The default value for this property for a newly created object is False.

This property is read-write.

SequenceNumber

Description

The sequence number is used when returning a collection. The default order of the returned PageletCategory objects is based on the sequence number. Use this property to reorder PageletCategory objects.

If there are duplicates in the sequence number, the PageletCategory objects are returned in alphabetical order.

The length of this property is 4 characters.

This property is read-write.

PageletCategory Collection

The PageletCategory Collection object provides access to a collection of PageletCategory objects in a PageletCategory object.

The PageletCategory Collection is instantiated from the PageletCategories PortalRegistry property.

See Chapter 21, PortalRegistry Classes , PageletCategories, page 1134.

PageletCategory Collection Methods

In this section, we discuss the PageletCategory collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*PageletCategoryName*)

Description

The **DeleteItem** method deletes the PageletCategory object identified by *PageletCategoryName* from the database.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for deletion, then actually deleted later. The item is deleted from the database *as soon as* the method is executed.

Parameters

Parameter	Description
PageletCategoryName	Specify the name of a PageletCategory existing in the PageletCategory collection.

Returns

A Boolean value: True if the PageletCategory was deleted, False otherwise.

Example

```
If Not &MyPageletCategoryColl.DeleteItem("MYPAGELETCATEGORY") Then
   /* PageletCategory not deleted. Do error checking */
End-If;
```

First

Syntax

First()

Description

The **First** method returns the first PageletCategory object in the PageletCategory collection.

Parameters

None.

Returns

PageletCategory object.

Example

```
&MyPageletCategory = &MyCollection.First();
```

InsertItem

Syntax

InsertItem(*PageletCategoryName, Label*)

Description

The **InsertItem** method inserts the PageletCategory object identified by *PageletCategoryName* from the PageletCategory Collection. You must specify both a name and a label for each PageletCategory. This method returns a reference to the new PageletCategory object. You must specify a unique *PageletCategoryName*, or you receive an error.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for insertion, then actually inserted later. The item is inserted into the database *as soon as* the method is executed.

Parameters

Parameter	Description
PageletCategoryName	Specify the name of a PageletCategory existing in the PageletCategory collection. This parameter takes a string value.
Label	Specify a label for the new PageletCategory. This parameter takes a string value.

Returns

A reference to the new PageletCategory object if the method executed successfully, False otherwise.

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the PageletCategory object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing PageletCategory within the PageletCategory collection. If you specify an invalid name, the object is NULL. You must specify a name, not a label.

Returns

A PageletCategory object if successful, NULL otherwise.

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next PageletCategory object in the PageletCategory collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A PageletCategory object.

PageletCategory Collection Property

In this section, we discuss the PageletCategory collection properties. The properties are discussed in alphabetical order.

Count

Description

This property returns the number of PageletCategory objects in the PageletCategory Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Pagelet Class

A pagelet represents a URL that renders a section of content for a homepage. Pagelets are assigned to specific categories.

A pagelet is instantiated from the First, InsertItem, ItemByName, and Next Pagelet Collection methods.

See Chapter 21, PortalRegistry Classes , Pagelet Collection, page 1224.

Pagelet Class Method

In this section, we discuss the Save method.

Save

Syntax

Save()

Description

The **Save** method saves any changes you made to the Pagelet, for example, a changed description or sequence number.

Parameters

None.

Returns

A Boolean value: True if the Pagelet object is successfully saved, False otherwise.

Example

```
If Not(&MyPagelet.Save()) Then
   /* do error checking */
End-If;
```

See Also

Chapter 21, PortalRegistry Classes , Save, page 1131 PortalRegistry method
Chapter 21, PortalRegistry Classes , Save, page 1161 ContentReference method
Chapter 21, PortalRegistry Classes , Save, page 1210 PageletCategory method

Pagelet Class Properties

In this section, we discuss the Pagelet class properties. The properties are discussed in alphabetical order.

Attributes

Description

This property returns an Attribute Collection containing the AttributeValues for this pagelet object.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , Attribute Collection, page 1178.

Author

Description

This property returns the author (PeopleSoft user ID) for this pagelet object as a string.

This property is read-only.

AuthorAccess

Description

This property specifies whether the author of the pagelet has access to the pagelet. This property takes a Boolean value. The default value for this property is True.

This property is read-write.

Authorized

Description

This property specifies whether the user is authorized to view this pagelet.

The initial value of this property depends on the other permission properties (PublicAccess and AuthorAccess) and the permission list values in the PermissionValue object associated with this pagelet.

This property is read-only.

CascadedPermissions

Description

This property returns a PermissionValue collection. This collection contains the value of the permissions for all the parent objects (up to the root folder). To determine only the permissions of the object use the Permission property instead.

Note. You *cannot* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can add values only to the collection returned by the Permissions property.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , PermissionValue Collection, page 1182, Chapter 21, PortalRegistry Classes , Permissions, page 1166.

ContentProvider

Description

This property returns or sets the name of the node for the pagelet as a string.

If no node was specified when the pagelet was created, this property returns Null.

This property is read-write.

CreationDate

Description

This property returns the creation date for this pagelet object as a string.

This property is read-only.

DefaultColumn

Description

This property returns or sets the default column this pagelet displays in. This property takes a number value.

The default value for this property is 1.

This property is read-write.

Description

Description

This property returns or sets the description for this pagelet object as a string.

The length of this property is 256 characters.

This property is read-write.

EditPageContentProvider

Description

This is the node part of the URL to the page that enables a user to personalize a page.
The default value is an empty string.

This property is read-write.

EditPageQueryString

Description

This is the query string part of the URL to the page that enables a user to personalize a page.
The default value for this property is an empty string.

This property is read-write.

HelpID

Description

This property returns the help system identifier for this pagelet. It enables the help system to provide pagelet specific help. This property takes a string value.

This property is read-write.

IsHideMinimize

Description

This property specifies if the minimize image icon displays. This property takes a Boolean value: True, display the icon, False, hide the icon. The default value is True.

This property is read-write.

Label

Description

This property returns or sets the label for this pagelet object as a string.

The length of this property is 30 characters.

This property is translatable.

This property is read-write.

Name

Description

This property returns the name for this pagelet object as a string. The name is a unique identifier for each pagelet object.

Every pagelet name must be unique in the portal, not just in the parent folder.

This property is read-only.

OwnerId

Description

This property returns or sets the owner ID of the pagelet object as a string.

This property is read-write.

ParentName

Description

This property returns the parent folder name for this pagelet object as a string.

This property is read-only.

Permissions

Description

This property returns a PermissionValue Collection. This collection contains the value of the permissions for this pagelet. To determine the permissions for all the parent objects (up to the root folder) use the CascadedPermissions property.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , PermissionValue Collection, page 1182, Chapter 21, PortalRegistry Classes , CascadedPermissions, page 1163.

Product

Description

This property returns or sets the PeopleSoft product for this pagelet object as a string.

The length of this property is 4 characters.

This property is read-write.

PublicAccess

Description

This property indicates whether a pagelet is generally accessible, that is, if it will always be included in the general pagelet collection. This property takes a Boolean value.

The default value for this property is False.

This property is read-write.

QualifiedURL

Description

This property returns an absolute URL. If the pagelet has a node associated with it, the URI from the node is concatenated with the URL from the pagelet. If there is no ContentProvider, this property returns the text in the URL property.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , URL, page 1170.

SequenceNumber

Description

The sequence number is used when returning a collection. The default order of the returned pagelets is based on the sequence number. Use this property to reorder pagelets. This property takes a number value.

If there are duplicates in the sequence number, the pagelets are returned alphabetically.

The length of this property is 4 characters.

This property is read-write.

URL

Description

This property returns or sets the URL for this pagelet object as a string. The URL returns *exactly* as it appears in the database.

If you're setting a URL, you must use a unique URL.

The absolute URL, that is, the URL from the ContentProvider concatenated with this URL must be unique.

You receive an error if the URL you use is already registered.

To retrieve a qualified URL, that is, one that contains the node URI, use the AbsoluteContentURL property.

The length of this property depends on your system database limit for LONG fields.

The format of the value for this property depends on the setting of other properties.

This property is read-write.

URLType

Description

This property indicates what kind of PeopleCode definition is used to retrieve the content of the pagelet. This property takes a string value.

Values are:

URLType Value	Meaning
UEXT	URL points to a non-PeopleSoft (external) URL
UPGE	URL points to a component.
UGEN	URL points to a generic PeopleSoft URL
USCR	URL points to an iScript

UPGE is the default value on a new pagelet.

This property is read-write.

Pagelet Collection

The Pagelet Collection provides access to the collection of pagelets in a category.

The Pagelet Collection is instantiated from the Pagelets PageletCategory property.

See Chapter 21, PortalRegistry Classes , Pagelets, page 1213.

Pagelet Collection Methods

In this section, we discuss the Pagelet collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*PageletName*)

Description

The **DeleteItem** method deletes the pagelet object identified by *PageletName* from the pagelet Collection.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for deletion, then actually deleted later. The item is deleted from the database *as soon as* the method is executed.

Parameters

Parameter	Description
PageletName	Specify the name of a pagelet existing in the pagelet collection.

Returns

A Boolean value: True if the pagelet was deleted, False otherwise.

Example

```
If Not &MyColl.DeleteItem("Test_Pagelet") Then
   /* can't delete test data. Do error processing */
End-if;
```

First

Syntax

```
First()
```

Description

The **First** method returns the first pagelet object in the pagelet collection.

Parameters

None.

Returns

A reference to a pagelet object if successful, NULL otherwise.

Example

```
&MyCRef = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(PageletName, Label, NodeName, URL)
```

Description

The **InsertItem** method inserts the pagelet object identified by *PageletName* into the pagelet Collection.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for insertion, then actually inserted later. The item is inserted into the database *as soon as* the method is executed.

Parameters

Parameter	Description
PageletName	Specify the name of a new pagelet. This parameter takes a string value. If you specify a name that already exists in the collection, you receive an error.
Label	Specify the label for the new pagelet. This parameter takes a string value.
NodeName	Specify a node. This parameter takes a string value. If you specify a fully qualified URL, you can specify a Null (that is, two quotation marks with no space between them (""")).
URL	Specify a URL that contains the content. The format of this parameter depends on other properties, such as the type of pagelet, where the data is stored, and so on.

Returns

A reference to the new pagelet object if the method executed successfully, NULL otherwise.

See Also

Chapter 21, PortalRegistry Classes , Using Content References , page 1106.

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the pagelet object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing pagelet within the pagelet collection. If you specify an invalid name, the object will be NULL.

Returns

A reference to a pagelet object if successful, NULL otherwise.

Example

```
&MyPagelet = &CRefColl.ItemByName("PORTAL_ADMIN);
```

Next

Syntax

Next()

Description

The **Next** method returns the next pagelet object in the pagelet collection. You can only use this method after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a pagelet object if successful, NULL otherwise.

Example

```
&MyPagelet = &MyCollection.Next();
```

Pagelet Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of pagelet objects in the pagelet Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

UserHomepage Class

The UserHomepage represents a homepage for a user. It contains all the tabs and pagelets the user has selected for their homepage.

A UserHomepage is instantiated from the Homepage PortalRegistry property.

See Chapter 21, PortalRegistry Classes , Homepage, page 1133.

UserHomepage Class Method

In this section, we discuss the Save method.

Save

Syntax

Save()

Description

The **Save** method saves any changes you made to the UserHomepage, for example, a changed greeting.

Using this method also saves any changes you've made to any UserTab or SelectedPagelet objects associated with this UserHomepage.

Parameters

None.

Returns

A Boolean value: True if the UserHomepage object is successfully saved, False otherwise.

Example

```
If Not(&MyHP.Save()) Then
   /* do error checking */
End-If;
```

See Also

Chapter 21, PortalRegistry Classes , Save, page 1131 PortalRegistry class method
Chapter 21, PortalRegistry Classes , Save, page 1151 Folder class method

UserHomepage Class Properties

In this section, we discuss the UserHomepage properties. The properties are discussed in alphabetical order.

Greeting

Description

This property is a user-definable message that appears on their homepage when they login, such as "My Homepage" or "Welcome." This property takes a text string. The length of this character is 254.

This property is read-write.

UserId

Description

This property returns the UserId for this UserHomepage as a string.

This property is read-only.

UserTab

Description

This property returns a reference to a UserTab collection.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , UserTab Collection, page 1230.

UserTab Class

A UserTab is a personalized tab for a specific user. This user can potentially change the pagelets that appear on the tab, the name of the tab, and the order of the tab. All these changes can be allowed or denied on the TabDefinitions Class. The UserTab is used in place of the TabDefinition whenever a user has personalized a tab.

A UserTab object is instantiated from the First, ItemByName, InsertItem, or Next UserTab Collection methods.

See Chapter 21, PortalRegistry Classes , UserTab Collection, page 1230.

UserTab Class Properties

In this section, we discuss the UserTab class properties. The properties are discussed in alphabetical order.

ColumnLayout

Description

This property specifies how many columns the UserTab contains. This property takes a numeric value. The values are:

Value	Description
2	Two-column layout
3	Three-column layout. This is the default value.

This property is read-write.

Label

Description

This property returns or sets the label for this UserTab object as a string. This value is initially set to the Label of the TabDefinition.

The length of this property is 30 characters.

This property is translatable.

This property is read-write.

QualifiedURL

Description

This property returns an absolute URL as a string. This URL includes the URI from the TabDefinition's node and the URL that points to the actual content of the tab.

This property is read-only.

SelectedPagelets

Description

This property returns a SelectedPagelets Collection, containing the pagelets that have been selected by the user to appear on this tab.

This property is read-only.

See Also

Chapter 21, PortalRegistry Classes , SelectedPagelet Collection, page 1234.

SequenceNumber

Description

This property specifies the order that this tab should appear as on the homepage. This property takes a number value. This value is initially set by the referenced TabDefinition.

This property is read-write.

TabName

Description

This property returns the tab name as a sting.

This property is read-only.

UserTab Collection

The UserTab collection is the collection of personalized tabs for a user.

The UserTab Collection is instantiated from the UserTabs UserHomepage property.

See Chapter 21, PortalRegistry Classes , UserTab, page 1228.

UserTab Collection Methods

In this section, we discuss the UserTab collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

```
DeleteItem(UserTabName)
```

Description

The **DeleteItem** method deletes the UserTab object identified by *UserTabName* from the UserTab Collection.

This method is not executed automatically. It is executed only when the parent object is saved.

Parameters

Parameter	Description
UserTabName	Specify the name of a UserTab existing in the UserTab collection.

Returns

A Boolean value: True if the UserTab was deleted, False otherwise.

Example

```
If Not &MyColl.DeleteItem("Test_UserTab") Then
   /* can't delete test data. Do error processing */
End-if;
```

First

Syntax

```
First()
```

Description

The **First** method returns the first UserTab object in the UserTab collection.

Parameters

None.

Returns

A UserTab object.

Example

```
&MyUserTab = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(UserTabName)
```

Description

The **InsertItem** method inserts the UserTab object identified by *UserTabName* into the UserTab Collection.

This method is not executed automatically. It is executed only when the parent object is saved.

The new user tab must be unique within the portal.

Parameters

Parameter	Description
UserTabName	Specify the name of a new UserTab. This parameter takes a string value. If you specify a name that already exists in the collection, you get an error.

Returns

A reference to the new UserTab object if the method executed successfully, NULL otherwise.

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the UserTab object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing UserTab within the UserTab collection. If you specify an invalid name, the object is NULL.

Returns

A UserTab object if successful, NULL otherwise.

Example

```
&MyUserTab = &MyColl.ItemByName("PORTAL_ADMIN");
```

Next

Syntax

Next()

Description

The **Next** method returns the next UserTab object in the UserTab collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A UserTab object.

Example

```
&MyUserTab = &MyCollection.Next();
```

UserTab Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of UserTab objects in the UserTab Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

SelectedPagelet Class

The SelectedPagelet class contains data on pagelets selected by the user that display on this tab.

A SelectedPagelet object is instantiated by the First, InsertItem, ItemByName, and Next SelectedPagelet Collection methods.

See Chapter 21, PortalRegistry Classes , SelectedPagelet Collection, page 1234.

SelectedPagelet Class Properties

In this section, we discuss the SelectedPagelet class properties. The properties are discussed in alphabetical order.

CategoryName

Description

This property returns the category of the pagelet as a string.

This property is read-only.

Column

Description

This property returns the number of the column the pagelet displays in, as a number.

The values are:

Value	Description
1	First column
2	Second column
3	Third column

This property is read-write.

IsMinimized

Description

This property specifies whether the pagelet is displayed minimized. This property takes a Boolean value: True, the pagelet is minimized, False otherwise.

This property is read-write.

PageletName

Description

This property returns the name of the pagelet as a string.

This property is read-only.

Row

Description

This property specifies which row the pagelet is displayed in, as a number.

This property is read-write.

SelectedPagelet Collection

The SelectedPagelet collection contains a collection of all the selected pagelets for a tab.

A SelectedPagelet collection is instantiated by the SelectedPagelets UserTab property.

See Chapter 21, PortalRegistry Classes , SelectedPagelets, page 1230.

SelectedPagelet Collection Methods

In this section, we discuss the SelectedPagelet collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*SelectedPageletName*)

Description

The **DeleteItem** method deletes the SelectedPagelet object identified by *SelectedPageletName* from the SelectedPagelet Collection.

This method is not executed automatically. It is executed only when the parent object is saved.

Parameters

Parameter	Description
SelectedPageletName	Specify the name of a SelectedPagelet existing in the SelectedPagelet collection.

Returns

A Boolean value: True if the SelectedPagelet was deleted, False otherwise.

Example

```
If Not &MyColl.DeleteItem("Test_SelectedPagelet") Then
   /* can't delete test data. Do error processing */
End-if;
```

First

Syntax

First()

Description

The **First** method returns the first SelectedPagelet object in the SelectedPagelet collection.

Parameters

None.

Returns

A SelectedPagelet object.

Example

```
&MySelectedPagelet = &MyCollection.First();
```

InsertItem

Syntax

InsertItem(*SelectedPageletName, Column, Row*)

Description

The **InsertItem** method inserts the SelectedPagelet object identified by *SelectedPageletName* into the SelectedPagelet Collection.

This method is not executed automatically. It is executed only when the parent object is saved.

Parameters

Parameter	Description
SelectedPageletName	Specify the name of a new SelectedPagelet. This parameter takes a string value. If you specify a name that already exists in the collection, you get an error.
Column	Specify the column number for this pagelet.
Row	Specify the row number for this pagelet.

Returns

A reference to the new SelectedPagelet object if the method executed successfully, NULL otherwise.

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the SelectedPagelet object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing SelectedPagelet within the SelectedPagelet collection. If you specify an invalid name, the object is NULL.

Returns

A SelectedPagelet object if successful, NULL otherwise.

Example

```
&MySelectedPagelet = &MyColl.ItemByName("PORTAL_ADMIN");
```

Next

Syntax

Next()

Description

The **Next** method returns the next SelectedPagelet object in the SelectedPagelet collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A SelectedPagelet object.

Example

```
&MySelectedPagelet = &MyCollection.Next();
```

SelectedPagelet Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of SelectedPagelet objects in the SelectedPagelet Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Favorite Class

A favorite represents a content reference the user wants to keep a shortcut to. It contains the name of the content reference and the label the user wants to see displayed for this favorite. Labels must be unique in each favorite collection.

Use the **Save** method of the Favorite class to save any changes you made to any favorite objects.

A favorite object is instantiated by the First, InsertItem, ItemByName, and Next Favorite Collection methods.

See Chapter 21, PortalRegistry Classes , Favorite Collection, page 1239.

Favorite Class Properties

In this section, we discuss the Favorite class properties. The properties are discussed in alphabetical order.

CRefName

Description

This property returns the name of the content reference as a string.

This property is read-only.

Label

Description

This property specifies the description of the favorite that's displayed to the user. This label must be unique for the favorite collection.

This property is read-write.

QualifiedURL

Description

This property returns the qualified URL for the content reference based on the Node and the URL for the name of the content reference, as a string.

This property is read-only.

SequenceNumber

Description

This property returns the sequence number used to order favorites.

This property is read-write.

URL

Description

This property returns the relative URL for this favorite as a string.

For example, the following code:

```
&URL = &MyFavorite.URL;
```

could return the following string:

```
c/PORTAL_PERS_HOMEPAGE.PORTAL_HOMEPAGE.GBL?PORTALPARAM_PAGE=CONTENT&tab=⇒
DEFAULT&PortalFavorite=QEDMO
```

This property is read-write.

Favorite Collection

The favorite collection provides access to the favorites for a particular user.

A favorite collection is instantiated by the Favorites PortalRegistry property.

See Chapter 21, PortalRegistry Classes , Favorites, page 1133.

Favorite Collection Methods

In this section, we discuss the Favorite collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*FavoriteLabel*)

Description

The **DeleteItem** method deletes the Favorite object identified by *FavoriteLabel* from the Favorite Collection.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for deletion, then actually deleted later. The item is deleted from the database *as soon as* the method is executed.

Parameters

Parameter	Description
FavoriteLabel	Specify the label of a Favorite existing in the Favorite collection, that is, the text used to identify the favorite to the end-user.

Returns

A Boolean value: True if the Favorite was deleted, False otherwise.

Example

```
If Not &MyColl.DeleteItem("My Local Test") Then
   /* can't delete test data. Do error processing */
End-if;
```

First

Syntax

```
First()
```

Description

The **First** method returns the first Favorite object in the Favorite collection.

Parameters

None.

Returns

A Favorite object.

Example

```
&MyFav = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(FavoriteLabel, FavoriteName)
```

Description

The **InsertItem** method inserts the Favorite object identified by *FavoriteName* into the Favorite Collection.

Note. The PortalRegistry classes execute some methods "interactively", that is, as they happen. The item won't be marked for insertion, then actually inserted later. The item is inserted to the database *as soon as* the method is executed.

Parameters

Parameter	Description
FavoriteLabel	Specify a label for the new Favorite. This is the text by which the favorite is identified to the end-user.
FavoriteName	Specify the name of a new Favorite. This parameter takes a string value. If you specify a name that already exists in the collection, you get an error.

Returns

A reference to the new Favorite object if the method executed successfully, NULL otherwise.

ItemByLabel

Syntax

```
ItemByLabel(FavoriteLabel)
```

Description

The **ItemByLabel** method returns the Favorite object with the label *FavoriteLabel*. The label is the text used to identify the favorite to the end-user.

Parameters

Parameter	Description
FavoriteLabel	Specify the label of an existing Favorite within the Favorite collection. If you specify an invalid favorite, the object returned is Null.

Returns

A Favorite object if successful, Null otherwise.

Example

```
&MyFavorite = &MyColl.ItemByLabel("My Local Login");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next Favorite object in the Favorite collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A Favorite object.

Example

```
&MyFavorite = &MyCollection.Next();
```

Save

Syntax

```
Save()
```

Description

Use the **Save** method to save any changes you made to the Favorite collection.

Parameters

None.

Returns

A Boolean value: True, the collection saved successfully, False otherwise.

Favorite Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of Favorite objects in the Favorite Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count
```

PortalRegistry Classes Example

There are several actions you may want to perform using a PortalRegistry, such as adding a ContentProvider, adding a folder, changing permissions, and so on. The following examples discuss the most common tasks. In addition, there are example programs of accessing the PortalRegistry classes using language environments other than PeopleCode.

Changing PortalRegistry Properties

This example shows how to change the default template used by a PortalRegistry. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MySession;
Local ApiObject &MyPortal, &MyPortalColl;

/* Access the current session */

&MySession = %Session;
If NOT &MySession Then
   /* do error processing */
End-if;

&MyPortalColl = &MySession.FindPortalRegistries();

For &I = 1 to &MyPortalColl.Count
```

```
      &MyPortal = &MyPortalColl.Item(&I);

      If &MyPortal.DefaultTemplate = "HR99" Then
         &MyPortal.DefaultTemplate = "HR00"
         &MyPortal.Save();
      End-If

   End-For;
```

To change PortalRegistry properties:

1. Get a Session object.

   Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, enables you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.)

   ```
   &MySession = %Session;
   ```

2. Get a PortalRegistry collection.

   Use the FindPortalRegistries method with no parameters to return a collection of all the PortalRegistries because you want to check all the registries for the invalid template.

   ```
   &MyPortalColl = &MySession.FindPortalRegistries();
   ```

   Use the Count property on the collection to loop through all the registries.

   ```
   For &I = 1 to &MyPortalColl.Count
   ```

3. Get a PortalRegistry object.

   You can access a PortalRegistry object from the PortalRegistry collection using the Item method.

   ```
   &MyPortal = &MyPortalColl.Item(&I);
   ```

4. Check for the invalid template and correct if necessary.

   Use the DefaultTemplate property both to check for the invalid template name, and to set the correct template. Save the PortalRegistry when you make a correction.

   ```
   If &MyPortal.DefaultTemplate = "HR99" Then
      &MyPortal.DefaultTemplate = "HR00"
      &MyPortal.Save();
   End-If;
   ```

You may want to do error checking after you save the PortalRegistry.

Adding a ContentProvider

The following example adds a ContentProvider to an existing PortalRegistry. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MyPortal;
Local ApiObject &MyCPColl, &MyCProvider;

&MyPortal = %Session.GetPortalRegistry();
```

```
If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
   /* Do error handling */
End-if;

/* Add a ContentProvider */

&MyCPColl = &MyPortal.ContentProviders;

&MyCProvider = &MyCPColl.InsertItem("HRMS_00");

&MyCProvider.URI = "http://MYMACHINE103100/servlets/iclientservlet/HRMS/";
&MyCProvider.DefaultTemplate = "MYPORTAL_HRMS";
&MyCProvider.Description = "Updated Content Provider for HRMS";

If NOT(&MyPortal.Save()) Then
   &ErrorCol = &MySession.PSMessages;
   For &I = 1 to &ErrorCol.Count
      &Error = &ErrorCol.Item(&I);
   /* do error processing */
   End-For;
&ErrorCol.DeleteAll();
End-If;
```

To add a ContentProvider:

1. Get a Session object and a PortalRegistry.

   Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, enables you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.) In addition, you want to get a PortalRegistry. Using the GetPortalRegistry method returns a reference to an unpopulated PortalRegistry object.

   ```
   &MyPortal = %Session.GetPortalRegistry();
   ```

2. Open the PortalRegistry.

   After you get a PortalRegistry object, you want to open it, that is, populate it with data. Use the **Open** method to do this. The Open method returns a Boolean value, and this example uses that value to do error checking to make sure that the PortalRegistry is actually opened. In addition, the name of the PortalRegistry is kept in the record MYRECORD, in the field PORTAL_NAME.

   ```
   If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
      /* Do error handling */
   End-if;
   ```

3. Access the ContentProvider collection.

   You can only add a ContentProvider to a ContentProvider collection. So you must access a ContentProvider collection first, using the ContentProviders property on a PortalRegistry.

   ```
   &MyCPColl = &MyPortal.ContentProviders;
   ```

4. Add the ContentProvider.

   You must use the name of the ContentProvider with the InsertItem method. This example uses HRMS_00. This method does *not* execute automatically, that is, the ContentProvider isn't actually inserted into the database until the PortalRegistry is saved.

   ```
   &MyCProvider = &MyCPColl.InsertItem("HRMS_00");
   ```

5. Further define the ContentProvider.

   The URI of a ContentProvider is used in conjunction with content references to define the location of content. So you should define the URI. If a template isn't found for any of the content references on a page at runtime, the system next tries to use the DefaultTemplate defined for a ContentProvider, so you should define a default template.

   ```
   &MyCProvider.URI = "http://MYMACHINE103100/servlets/iclientservlet/HRMS/";
   &MyCProvider.DefaultTemplate = "MYPORTAL_HRMS";
   &MyCProvider.Description = "Updated Content Provider for HRMS";
   ```

6. Save the PortalRegistry and check for errors.

   Because there is no Save method with a ContentProvider, you must save the PortalRegistry to complete your changes.

   You can check if there were any errors using the PSMessages property on the session object.

   ```
   If NOT(&MyPortal.Save()) Then
      /* save didn't complete */
      &ErrorCol = &MySession.PSMessages;
      For &I = 1 to &ErrorCol.Count
         &Error = &ErrorCol.Item(&I);
         /* do error processing */
      End-For;
      &ErrorCol.DeleteAll();
   End-if;
   ```

   If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, delete it from the PSMessages collection.

   Note. If you've called the PortalRegistry API an Application Engine program, all errors are also logged in the Application Engine error log tables.

See Chapter 29, Session Class , Error Handling , page 1560.

Changing ContentProvider Properties

In the following example, you change the name of a DefaultTemplate for all ContentProviders that have a specific DefaultTemplate. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MyPortal;
Local ApiObject &MyCPColl, &MyCProvider;

&MyPortal = %Session.GetPortalRegistry();
```

```
If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
   /* Do error handling */
End-if;

/* Add a ContentProvider */

&MyCPColl = &MyPortal.ContentProviders;

&MyCProvider = &MyCPColl.First();
&MyCount = 1;

For &I = 1 to &MyCPColl.Count

/* Check template name and change if necessary */

   If &MyCProvider.DefaultTemplate = "INV_DEPT_SALES" Then
      &MyCProvider.DefaultTemplate = "UK_INV_DEPT_SALES";
   End-If;

/* If not the end of the collection, get next ContentProvider */

   If &MyCount <> &MyCPColl.Count
      &MyCount = &MyCount + 1;
      &MyCProvider = &MyCPColl.Next();
   End-If;

End-For;

If NOT(&MyPortal.Save()) Then
&ErrorCol = &MySession.PSMessages;
   For &I = 1 to &ErrorCol.Count
      &Error = &ErrorCol.Item(&I);
   /* do error processing */
   End-For;
&ErrorCol.DeleteAll();
End-If;
```

To change a ContentProvider:

1. Get a Session object and a PortalRegistry.

   Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, enables you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.) In addition, you want to get a PortalRegistry. Using the GetPortalRegistry method returns a reference to an unpopulated PortalRegistry object.

   ```
   &MyPortal = %Session.GetPortalRegistry();
   ```

2. Open the PortalRegistry.

After you get a PortalRegistry object, you want to open it, that is, populate it with data. Use the **Open** method to do this. The Open method returns a Boolean value, and this example uses that value to do error checking to make sure that the PortalRegistry is actually opened. In addition, the name of the PortalRegistry is kept in the record MYRECORD, in the field PORTAL_NAME.

```
If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
   /* Do error handling */
End-if;
```

3. Access the ContentProvider collection.

   You can add a ContentProvider only to a ContentProvider collection. So you must access a ContentProvider collection first, using the ContentProviders property on a PortalRegistry.

   ```
   &MyCPColl = &MyPortal.ContentProviders;
   ```

4. Get the first ContentProvider in the collection.

   This example loops through all the ContentProviders in the collection, starting with the first one, then using the **Next** method to access each one, to the end of the list. This example uses a second counter to determine the end of the list.

   ```
   &MyCProvider = &MyCPColl.First();
   &MyCount = 1;
   ```

5. Check the name of the DefaultTemplate and change it if necessary.

   This example uses the Count property with the ContentProvider collection to determine how many times to loop through. Then it checks the existing name of the DefaultTemplate. If it's the incorrect name, the name is changed to the correct name.

   ```
   For &I = 1 to &MyCPColl.Count

   /* Check template name and change if necessary */

      If &MyCProvider.DefaultTemplate = "INV_DEPT_SALES" Then
         &MyCProvider.DefaultTemplate = "UK_INV_DEPT_SALES";
      End-If;
   ```

6. Check the second counter to determine if at the end of the collection.

   If the program hasn't reached the end of the ContentProvider collection, access the next ContentProvider.

   ```
   If &MyCount <> &MyCPColl.Count
      &MyCount = &MyCount + 1;
      &MyCProvider = &MyCPColl.Next();
   End-If;
   ```

7. Save the PortalRegistry and check for errors.

   Because there is no Save method with a ContentProvider, you must save the PortalRegistry to complete your changes.

   You can check if there were any errors using the PSMessages property on the session object.

   ```
   If NOT(&MyPortal.Save()) Then
      /* save didn't complete */
   ```

```
   &ErrorCol = &MySession.PSMessages;
   For &I = 1 to &ErrorCol.Count
      &Error = &ErrorCol.Item(&I);
      /* do error processing */
   End-For;
   &ErrorCol.DeleteAll();
End-if;
```

If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, delete it from the PSMessages collection.

Note. If you've called the PortalRegistry API an Application Engine program, all errors are also logged in the Application Engine error log tables.

See Chapter 29, Session Class , Error Handling , page 1560.

Adding a Folder

The following example checks to see if a folder for a user exists. If it doesn't, it creates one.

The folder hierarchy for this example is as follows. The Users folder is where the portal stores things such as user homepages:

```
Root folder

  Users folder

     UserId1 folder

     UserId2 folder

     UserId3 folder

     . . .
```

The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MyPortal;
Local ApiObject &UserFldrColl, &UserFldr;

&MyPortal = %Session.GetPortalRegistry();

If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
   /* Do error handling */
End-if;

&UserFldrColl = &MyPortal.RootFolder.Folders.ItemByName("Users").Folders;

&UserFldr = &UserFldrColl.ItemByName(%UserId);

If &UserFldr = Null Then

/* add Folder */
```

```
   &UserFldr = &UserFldrColl.InsertItem(%UserId, %UserId);
   &UserFldr.Description = %UserId;

/* Set dates */

   &UserFldr.ValidFrom = %Date;
   &ToDate = AddToDate(%Date, 1, 1, 0);
   &UserFldr.ValidTo = &ToDate;

/* Set properties */

   &UserFldr.PublicAccess = True;

/* save the folder */

   &UserFldr.Save();

End-If;
```

To add a folder:

1. Get a Session object and a PortalRegistry.

   Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, enables you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.) In addition, you want to get a PortalRegistry. Using the GetPortalRegistry method returns a reference to an unpopulated PortalRegistry object.

   ```
   &MyPortal = %Session.GetPortalRegistry();
   ```

2. Open the PortalRegistry.

   After you get a PortalRegistry object, you want to open it, that is, populate it with data. Use the **Open** method to do this. The Open method returns a Boolean value, and this example uses that value to do error checking to make sure that the PortalRegistry is actually opened. In addition, the name of the PortalRegistry is kept in the record MYRECORD, in the field PORTAL_NAME.

   ```
   If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
      /* Do error handling */
   End-if;
   ```

3. Access the User folder collection.

   One of the strengths of dot notation is being able to string together many methods and properties into a single line of code. The result of this single line of code is to return a reference to the folder collection within the Users folder.

   ```
   &UserFldrColl = &MyPortal.RootFolder.Folders.ItemByName("Users").Folders;
   ```

   Note that in this kind of code, you must access the RootFolder for the PortalRegistry before you can access the folders collection.

If there were additional folders in your hierarchy, you could continue in the same way, using ItemByName and Folders.

4. Check if the user already has a folder.

   The system variable %UserId returns the user ID of the current user. Then this example uses the ItemByName method on the folder collection to see if the user already has a folder. This assumes that the user folders are names according to the UserId. The ItemByName method returns a reference to the folder if it exists. If the folder does not exist, ItemByName returns NULL.

   ```
   &UserFldr = &UserFldrColl.ItemByName(%UserId);

   If &UserFldr = Null Then
   ```

5. Add a folder if one doesn't exist.

   Use the UserId as the name of the folder, as well as for the label and the description.

   ```
   &UserFldr = &UserFldrColl.InsertItem(%UserId, %UserId);
   &UserFldr.Description = %UserId;
   ```

6. Set the ValidFrom and ValidTo dates.

   For this example, the folder should be accessible from the date it's created, so the example sets the ValidFrom property to be today's date.

   When you first create a folder, the ValidTo date, that is, the date that this folder "expires", is set to null, that is an empty string. This means it never expires. In this example, we set the ValidTo date to one year and one day after the current date.

   ```
   &UserFldr.ValidFrom = %Date;
    &ToDate = AddToDate(%Date, 1, 1, 0);
    &UserFldr.ValidTo = &ToDate;
   ```

7. Set permissions for the folder.

   There are several properties that work together to determine the access of a folder or content reference. This example sets the PublicAccess property to True, which means that everyone has access to it, and it's included in all collections of folders.

   ```
   &UserFldr.PublicAccess = True;
   ```

   This example sets only the permission properties on the folder. It doesn't set all the possible permissions, such as the PermissionValue objects for the folder.

8. Save the folder.

   After you have finished your changes to the folder, save it.

   ```
   &UserFldr.Save();
   ```

   You may want to check for errors after you save each folder.

See Chapter 21, PortalRegistry Classes , Using Security , page 1097.

Adding a Content Reference

The following code example adds a content reference that's a target component to the Approve Expenses folder.

The folder hierarchy for this example is:

```
Root folder

  Expenses folder

     Create Expenses folder

     Review Expenses folder

     Approve Expenses folder
```

The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MyPortal;
Local ApiObject &CRef, &CRefColl;
Local ApiObject &Fldr;

&MyPortal = %Session.GetPortalRegistry();

If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
   /* Do error handling */
End-if;

&Fldr = &MyPortal.RootFolder.Folders.ItemByName("Expenses").Folders.ItemByName⇒
("Approve Expenses");

/* add content reference */

&CRef = &Fldr.ContentRefs.InsertItem(&CRefname, MYCP, MYRECORD.MYURL);
   &CRef.Description = &CRefname;

/* Set dates */
&CRef.ValidFrom = %Date;
&ToDate = AddToDate(%Date, 1, 1, 0);
&CRef.ValidTo = &ToDate;

/* Set content reference  types */
&CRef.UsageType = "TARG";   /* Target content reference */
&CRef.URLType = "UPGE";     /* Component type of content reference */
&CRef.StorageType = "RMTE";  /* Template is remote. */
&CRef.Template = "EXPENSES_TEMPLATE";      /* name of template */
&CRef.TemplateType = "HTML";   /* type of Template */
/* Set URL */
&CRef.URL = "ICType=Panel&Menu=MANAGE_EXPENSES&Market=GBL&Component=APPROVE_FINAL"

/* Save the content reference */

&CRef.Save();
```

To add a content reference:

1. Get a Session object and a PortalRegistry.

Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, enables you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.) In addition, you want to get a PortalRegistry. Using the GetPortalRegistry method returns a reference to an unpopulated PortalRegistry object.

```
&MyPortal = %Session.GetPortalRegistry();
```

2. Open the PortalRegistry.

   After you get a PortalRegistry object, you want to open it, that is, populate it with data. Use the **Open** method to do this. The Open method returns a Boolean value, and this example uses that value to do error checking to make sure that the PortalRegistry is actually opened. In addition, the name of the PortalRegistry is kept in the record MYRECORD, in the field PORTAL_NAME.

```
If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
   /* Do error handling */
End-if;
```

3. Access the folder collection.

   One of the strengths of dot notation is being able to string together many methods and properties into a single line of code. The result of this single line of code is to return a reference to the folder collection within the folder.

```
&Fldr = &MyPortal.RootFolder.Folders.ItemByName("Expenses").Folders.ItemByName⇒
("Approve Expenses");
```

   The Adding a Folder example has more explanation on the break down of this line of code.

4. Add the content reference.

   You can only add content references from the collection of content references for the folder. After you have the content reference collection, use the InsertItem method to add the content reference. The ContentProvider for this content reference is named MYCP. The URL is stored in the record MYRECORD in the field MYURL.

```
&CRef = &Fldr.ContentRefs.InsertItem(&CRefname, MYCP, MYRECORD.MYURL);
&CRef.Description = &CRefname;
```

   Note that the InsertItem method adds the content reference to the database as soon as it's executed. You could check for errors at this point to see if the content reference was added correctly.

5. Set the ValidFrom and ValidTo dates.

   For this example, the content reference should be accessible from the date it's created, so the example sets the ValidFrom property to be today's date.

   When you first create a content reference, the ValidTo date, that is, the date that this content reference "expires", is set to null, that is an empty string. This means it never expires. In this example, we set the ValidTo date to one year and one day after the current date.

```
&CRef.ValidFrom = %Date;
&ToDate = AddToDate(%Date, 1, 1, 0);
&CRef.ValidTo = &ToDate;
```

6. Set the type parameters.

Many content reference properties work together to set the type of content reference.

In this example, the content reference is a target. It's a component type of content reference, which means it's a PeopleSoft definition. The template to be used for displaying this content reference is stored remotely, its name is EXPENSES_TEMPLATE, and it's an HTML template.

```
&CRef.UsageType = "TARG"; /* Target content reference */
&CRef.URLType = "UPGE"; /* Component type of content reference */
&CRef.StorageType = "RMTE"; /* Template is remote. */
&CRef.Template = "EXPENSES_TEMPLATE"; /* name of template */
&CRef.TemplateType = "HTML"; /* type of Template */
```

7. Set the URL.

   The URL property of the content reference indicates where the content actually is. As this content reference is referencing a page, the URL has a specific format to indicate which page.

   ```
   &CRef.URL = "ICType=Panel&Menu=MANAGE_EXPENSES&Market=GBL&Component=APPROVE_FINAL"
   ```

8. Save the content reference.

   After you have finished adding the content reference, you should save it.

   ```
   &CRef.Save();
   ```

   You may want to check for errors after you save each content reference.

See Chapter 21, PortalRegistry Classes , Adding a Folder, page 1248.

Setting Permissions Using the PermissionValue Object

The following code example adds permissions for a folder through the PermissionValue object.

The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MyPortal;
Local ApiObject &UserFldr;
Local ApiObject &PermV, &PermVColl;

&MyPortal = %Session.GetPortalRegistry();

If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
   /* Do error handling */
End-if;

&UserFldr = &MyPortal.RootFolder.Folders.ItemByName("Users");

&CascadedColl = &UserFldr.CascadedPermissions;

&PermV = &CascadedColl.ItemByName("VENDOR1");

If NOT &PermV Then
   &PermVColl = &UserFldr.Permissions;
   &PermVColl.InsertItem("VENDOR1");
   &UserFldr.Save();
```

```
End-If;
```

To set permissions for a folder, using the PermissionValue object:

1. Get a Session object and a PortalRegistry.

   Before you can access a PortalRegistry object, you must get a session object. The session controls access to the registry, provides error tracing, enables you to set the runtime environment, and so on. Because you want to use the existing session, use the %Session system variable (instead of the GetSession function.) In addition, you want to get a PortalRegistry. Using the GetPortalRegistry method returns a reference to an unpopulated PortalRegistry object.

   ```
   &MyPortal = %Session.GetPortalRegistry();
   ```

2. Open the PortalRegistry.

   After you get a PortalRegistry object, you want to open it, that is, populate it with data. Use the **Open** method to do this. The Open method returns a Boolean value, and this example uses that value to do error checking to make sure that the PortalRegistry is actually opened. In addition, the name of the PortalRegistry is kept in the record MYRECORD, in the field PORTAL_NAME.

   ```
   If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
      /* Do error handling */
   End-if;
   ```

3. Access the User folder.

   One of the strengths of dot notation is being able to string together many methods and properties into a single line of code. The result of this single line of code is to return a reference to the folder named Users.

   ```
   &UserFldr = &MyPortal.RootFolder.Folders.ItemByName("Users");
   ```

4. Access the entire, cascaded PermissionValue collection.

   There are two types of PermissionValue collections you can access for a folder.

   - One contains only the permission list values that are set at that folder. This collection is returned with the Permissions property.
   - The other contains all the permission list values for folder, that is, it contains any permission list values set in any parent folder and cascaded. This collection is returned with the CascadedPermissions property.

Note. You *cannot* add any PermissionValue objects to a collection returned by the CascadedPermissions property. You can add values only to the collection returned by the Permissions property.

This example uses the CascadedPermissions collection to check if the permission exists, because we don't want to add a permission if it already exists. Then it uses the Permissions collection to add the value.

```
&CascadedColl = &UserFldr.CascadedPermissions;
```

1. Check to see if the permission list value exits.

   Before adding the new PermissionValue object, you want to make sure one by that name doesn't already exist.

   ```
   &PermV = &CascadedColl.ItemByName("VENDOR1");
   ```

2. If the PermissionValue doesn't exist, add it.

The ItemByName returns a reference to a PermissionValue object if it exists, or NULL if it doesn't. So this example checks for NULL, and adds the PermissionValue if it doesn't exist.

Notice that this example is *not* changing the **Cascaded** property with the PermissionValue object. The default value for a new PermissionValue object is False. As this example does not want to cascade the permissions for this permission list, it retains the default value.

In addition, if the PermissionValue object is added, the folder is saved. The InsertItem method executes only after the parent object, the folder, is saved.

```
If NOT &PermV Then
   &PermVColl = &UserFldr.Permissions;
   &PermVColl.InsertItem("VENDOR1");
   &UserFldr.Save();
End-If;
```

You may want to check for errors after you save the folder.

See Chapter 21, PortalRegistry Classes , Permissions, page 1156 and Chapter 21, PortalRegistry Classes , CascadedPermissions, page 1153.

See Chapter 21, PortalRegistry Classes , Using PermissionValue and Cascading Permissions , page 1098.

Using Attributes

The following example uses attributes for a folder to determine what is displayed to an end-user. The end-user still has access to the content, however, it isn't displayed. This program doesn't discuss how to hide or display content: it just shows the function used to determine if a folder should be displayed.

The following is the complete code sample: the steps explain each line.

```
Function checkVisible(&FolderId As ApiObject) Returns Boolean
   &colAttrib = &FolderId.Attributes;

   If &colAttrib.ItemByName("PORTAL_HIDE_FROM_NAV") <> Null Then
      If &colAttrib.ItemByName("PORTAL_HIDE_FROM_NAV").value = "TRUE" Then
             &display = False;
      End-If;
   End-If;

    Return &display;

End-Function;
```

To use attributes for a folder:

1. Access the folder attribute collection.

   A reference to a folder is passed into the function. Then the example uses the Attributes property on the folder to access the attribute collection for the folder.

   ```
   &colAttrib = &FolderId.Attributes;
   ```

2. Check to see if the folder should be hidden.

This step is actually two checks. The first check is to see if there is an attribute with the folder called PORTAL_HIDE_FROM_NAV. If the folder has such an attribute, the second check determines the value of this attribute. If both are true, the folder should be hidden, so the variable &display is set to True, and returned.

```
If &colAttrib.ItemByName("PORTAL_HIDE_FROM_NAV") <> Null Then
      If &colAttrib.ItemByName("PORTAL_HIDE_FROM_NAV").value = "TRUE" Then
            &display = False;
      End-If;
   End-If;

    Return &display;
```

Visual Basic Example

The following is an example referencing a PortalRegistry object using Visual Basic.

```
Dim oCref As ContentRef
Dim oCrefColl As ContentRefCollection

Set oSession = CreateObject("PeopleSoft.Session")
iResult = oSession.Connect(1, AppServStr, OperID, OperPasswd, 0)

Set oPortal = oSession.GetPortalRegistry
iResult = oPortal.Open("PORTAL")

Set oCrefColl = oPortal.RootFolder.ContentRefs
Set oCref = oCrefColl.InsertItem("papi012")
iResult = oPortal.Close

iResult = oPortal.Open("PORTAL")

Set oCref = oPortal.FindCRefByName("papi012")

oPortal.Close
oSession.Disconnect
Exit Sub
```

C/C++ Example

The following is a C/C++ example.

```
/*********************************************************************
*                                                                    *
* psportal_test.cpp                                                  *
*                                                                    *
*********************************************************************
*                                                                    *
*               Confidentiality Information:                         *
*                                                                    *
* This module is the confidential and proprietary information of
```

```
* PeopleSoft, Inc.; it is not to be copied, reproduced, or
* transmitted in any form, by any means, in whole or in part,
* nor is it to be used for any purpose other than that for which it
* is expressly provided without the written
* permission of PeopleSoft.
*                                                                      *
* Copyright (c) 1988-1999 PeopleSoft, Inc.  All Rights Reserved.       *
*                                                                      *
************************************************************************
*                                                                      *
* SourceSafe Information:                                              *
*                                                                      *
* $Logfile::                                                          $*
* $Revision::                                                         $*
* $Date::                                                             $*
*                                                                       *
************************************************************************/
/************************************************************************
*       includes                                                       *
************************************************************************/

#ifdef PS_WIN32
#include "stdafx.h"
#endif

#include "bcdef.h"
#include "apiadapterdef.h"
#include "peoplesoft_peoplesoft_i.h"

#include <stdio.h>
#include <iostream.h>
#include <wchar.h>

/************************************************************************
*       general defines and macros                                     *
************************************************************************/

/************************************************************************
*       globals                                                        *
************************************************************************/
HPSAPI_SESSION                                          hSession;


/************************************************************************
*       function prototypes                                            *
************************************************************************/
void DisconnectSession();
void GetFolder(HPSAPI_SESSION hSession, LPTSTR pszPortalName);
void OutError(HPSAPI_SESSION hSession);
```

```
/*********************************************************************
* Function:     main                                                 *
*                                                                    *
* Description:                                                       *
*                                                                    *
* Returns:                                                           *
*********************************************************************/
void main(int argc, char* argv[])
{
    // Declare variables
    PSAPIVARBLOB    blobExtra;
    TCHAR szServer[80] = _T("//LCHE0110:900");
    TCHAR szUserid[30] = _T("PTMO");
    TCHAR szUserPswd[80] = _T("PT");
    TCHAR szPortalName[80] = _T("PORTAL");

    memset(&blobExtra, 0, sizeof(PSAPIVARBLOB));

    // Establish a PeopleSoft Session.
   HPSAPI_SESSION hSession = PeopleSoft_Session_Create();

    if (PeopleSoft_Session_Connect(hSession, 1, szServer, szUserid, szUserPswd,⇒
 blobExtra))
    {
        GetFolder(hSession, szPortalName);
   }
   else
    {
      // Connect to AppServer Failed. Error out.
        if (PeopleSoft_Session_GetErrorPending(hSession))
      {
         wprintf(L"\nConnect to AppServer Failed.\n");
         OutError(hSession);
      }
      else
         wprintf(L"No error pending from session.\n");
    }
    DisconnectSession();
}


/*********************************************************************
*       function implementations                                     *
*********************************************************************/
/*********************************************************************
* Function:     DisconnectSession                                    *
*                                                                    *
* Description:  Disconnects the Session object.                      *
*                                                                    *
* Returns:      None                                                 *
```

```
*********************************************************************/
void DisconnectSession()
{
    // Disconnect current session to free memory and session variables.
    if (hSession)
    {
        if (PeopleSoft_Session_Disconnect(hSession))
        {
            PeopleSoft_Session_Release(hSession);
        }
        else
        {
            wprintf(L"Disconnect to AppServer Failed.\n");
        }
    }
}


/*********************************************************************
* Function:     GetFolder                                            *
*                                                                    *
* Description:  Get root folder and display folder name              *
*                                                                    *
* Returns:      None                                                 *
*********************************************************************/
void GetFolder(HPSAPI_SESSION hSession, LPTSTR pszPortalName)
{
    LPTSTR                  szStr;
    HPSAPI_PORTALREGISTRY   hPortal;
    HPSAPI_FOLDER           hRootFolder;

    hPortal=(HPSAPI_PORTALREGISTRY) PeopleSoft_Session_GetPortalRegistry(hSession);
    PortalRegistry_PortalRegistry_Open(hPortal, pszPortalName);
    if (PortalRegistry_PortalRegistry_Open(hPortal, pszPortalName) == false)
        _tprintf(_T("Open portal failed\n"));
    else
    {
        hRootFolder = PortalRegistry_PortalRegistry_GetRootFolder(hPortal);
        szStr = PortalRegistry_Folder_GetName(hRootFolder);
        _tprintf(_T("root folder name = %s\n"), szStr);
    }
}


/*********************************************************************
* Function:     OutError                                             *
*                                                                    *
* Description:  Output error message from session psMessage object   *
*                                                                    *
* Returns:      None                                                 *
```

```
*********************************************************************/
void OutError(HPSAPI_SESSION hSession)
{
    LPTSTR                  szStr;
    HPSAPI_PSMESSAGECOLLECTION hMsgColl;
   HPSAPI_PSMESSAGE hMsg;


    hMsgColl=(HPSAPI_PSMESSAGECOLLECTION) PeopleSoft_Session_GetPSMessages(h⇒
Session);

   wprintf(L"Get psMessage collection ok.\n");

   int i;
   PSI32 count;
   count=(PSI32) PeopleSoft_PSMessageCollection_GetCount;
   wprintf(L"Count= %d\n", count);

   for (i=0; i<count; i++)
   {
      hMsg=(HPSAPI_PSMESSAGE) PeopleSoft_PSMessageCollection_Item(hMsgColl, i);
      szStr=PeopleSoft_PSMessage_GetText(hMsg);
      wprintf(L"\nMessage from session: %s\n",szStr);
   }

}
```

CHAPTER 22

PostReport Class

This chapter provides an overview of PostReport class and discusses the following topics:

- Data type of a PostReport object
- Scope of a PostReport object
- Determining the distribution list
- PostReport class built-in functions
- PostReport class properties
- PostReport examples

See Also

PostReport Class Overview

The PostReport class provides properties and method for posting reports to the Report Repository. This class enables you to create reports outside of Process Scheduler and have them posted in the Report Repository.

The request to post reports creates an entry to the Report Manager table. The Distribution Agent polls the Report Manager table for any new requests and proceeds with any request it finds by transferring the files to the repository server. The report can then be accessed from the Report Manager after the Distribution Agent successfully processes the post request.

See Also

Determining the Distribution List

Use the **AddDistributionOption** method to authorize users to view the report from the Report Manager. This function enables you to grant access to the report either specifying a User ID or Role. However, if no distribution list is specified in the request, the class queries the system for the distribution list as follows:

1. If the Process Instance is specified, the system checks for the distribution list in the Process Request table. The system assumes in this case that the request was scheduled through the Process Request Dialog, but was processed by a third-party scheduler.
2. If a Process Name and Process Type properties were specified, the system checks if a distribution list was predefined in the Process Definition table.

3. If no list was found using the above queries, the user who submitted the request has sole access to the report.

Data Type of a PostReport Object

A PostReport object is declared using the PostReport data type. For example:

```
Local PostReport &POSTRQST;
```

Scope of a PostReport Object

A PostReport object can only be instantiated from PeopleCode. However, if you need to access the Process Schedule Manger from outside of PeopleCode, like in a Visual Basic program, you can use the Process Schedule Manager API.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

See Also

PeopleSoft Process Scheduler

PostReport Class Built-in Function

Chapter 1, PeopleCode Built-in Functions, SetPostReport , page 425

PostReport Class Methods

In this section, we discuss the PostReport class methods. The methods are discussed in alphabetical order.

AddDistributionOption

Syntax

AddDistributionOption(*DistIdType, DistId*)

Description

Use the **AddDistributionOption** method to specify the users authorized to view the report once it is available in the Report Manager.

Parameters

Parameter	Description
DistIdType	Specify the distribution ID type as a string. This identifies if the value passed in the DistID is either a user or role. Values for this parameter are: • User • Role
DistID	Specify the distribution ID as a string.

Returns

Returns a number: 0 if successful, 1 if the system detected an error in the parameter passed.

Example

The following example grants access to the VP1 user and all users associated with the MANAGERS role.

```
&POSTRQST.AddDistributionOption("User", "QEDMO" );
&POSTRQST.AddDistributionOption("Role", "MANAGERS");
```

Put

Syntax

```
Put()
```

Description

The **Put** method inserts a request in the Report Manager table which runs according to the values in the properties of the PostReport object. In order to successfully schedule a process or job, certain properties are required.

Parameters

None.

Returns

None. If you want to verify that the method executed successfully, check the value of the Status property.

Example

```
&POSTRQST.Put();
&RPTINSTANCE = &POSTRQST.ReportId;
If (&RPTINSTANCE > 0) Then
      MessageBox(0, "", 63, 119, "Successfully processed request with Rpt. ID %1⇒
 for Process %2 to post from directory %3", &RPTINSTANCE, &POSTRQST.ProcessName,⇒
 &POSTRQST.SourceReportPath);
   Else
      MessageBox(0, "", 63, 122, "Not successful for process request for Process⇒
 %1 to post from directory %2", &POSTRQST.ProcessName, &POSTRQST.SourceReportPath);
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, SetPostReport , page 425

PostReport Class Properties

In this section, we discuss the PostReport class properties. The properties are discussed in alphabetical order.

ExpirationDate

Description

This property specifies the date the report will be deleted from the Report Manager table. If the expiration date is not specified, the system calculates the expiration date based on the Retention Days specified in the Process Scheduler's System Settings.

This property is read-write.

Example

```
&MYRQST.ExpirationDate = "2003/01/01";
```

See Also

ReportFolder

Description

This property specifies the name of the report folder the entry is located under when viewing the report from the List or Explorer page found in the Report Manager component. If no folder is specified, the report is assigned to system default folder as defined in Process Scheduler's System Settings.

This property is read-write.

See Also

ReportId

Description

This property is a system-generated identification number. The system assigns a ReportId after the Put method processed the request.

This property is read-write.

Example

```
&MYRQST.Put();
if &MYRQST.ReportId = 0 then
     MessageBox(o, "", 63, 119, "Sucessfully processed request with Rpt. ID %1",⇒
 &MYRQST.ReportId);
Else
    /* do error processing */
End-If;
```

OutDestFormat

Description

This property specifies the output format of the file that is posted to the Report Repository. If format is not specified, the system determines the format based on the extension of the file being posted to the Report Repository.

This property is read-write.

Example

```
&MYRQST.OutDestFormat = "PDF";
```

See Also

Chapter 23, ProcessRequest Class , Values for Output Type and Format , page 1274

ProcessInstance

Description

This property specifies the instance number assigned to request. This property is only required if the report was initially submitted through the Process Request Dialog, but was processed by a third-party scheduler.

This property is read-write.

See Also

Chapter 22, PostReport Class , Determining the Distribution List , page 1261

ProcessName

Description

This property specifies the name of a predefined process as a string.

This property is read-write.

Example

```
&MYRQST.ProcessName = "XRFWIN";
```

See Also

Chapter 22, PostReport Class , ProcessType, page 1266
Chapter 23, ProcessRequest Class , RunControlID, page 1297

ProcessType

Description

This property specifies the name of a predefined process type as a string.

This property is read-write.

Example

Note that spaces are included in the string for ProcessType.

```
&MYRQST.ProcessType = "Application Engine";
```

See Also

Chapter 22, PostReport Class , ProcessName, page 1265
Chapter 23, ProcessRequest Class , RunControlID, page 1297

ReportDescr

Description

This property specifies the description as a string.

If the ReportDescr property is not specified, and both Process Name and Process Type are not null, the description is extracted from the Process Definition table.

This property is read-write.

Example

```
&MYRQST.ReportDescr = "Panel Cross Reference Report";
```

ReportFolder

Description

This property specifies the name of the report folder as a string. If you specify a folder, the folder that you specify must have already been defined using Process Scheduler’s Report Folders Administration..

This property is read-write.

See Also

ServerName

Description

This property specifies the Process Scheduler Server name that posts the report. This property takes a string value.

This property is read-write.

Example

```
&MYRQST.ServerName = "PSNT";
```

See Also

PostReport Class Examples

The following example posts files found in the directory ‘c:\temp\SQRDIR’ and can be accessed by VP1 and users in the MANAGERS role.

```
Local PostReport &RPTINFO;
   Local number &RPTINSTANCE

   /*********************************************************************
   * Construct a PostReport Object.                                     *
   *********************************************************************/
   &RPTINFO = SetPostReport();

   &RPTINFO.ProcessName = "GLS7009";
   &RPTINFO.ProcessType = "SQR Report";
   &RPTINFO.ReportFolder = "Financial";
   &RPTINFO.SourceReportPath = "c:\temp\SQRDIR";
   &RPTINFO.ExpirationDate = "2005/01/01";
   &RPTINFO.ReportDescr = "Journal Posting Summary Report";
   &RPTINFO.ServerName = "PSNT";

   &RPTINFO.AddDistributionOption("USER", "VP1");
   &RPTINFO.AddDistributionOption("ROLE", "MANAGERS");

   &RPTINFO.Put();
   &RPTINSTANCE = &RPTINFO.ReportId;

   If (&RPTINSTANCE > 0) Then
      MessageBox(0, "", 63, 119, "Successfully processed request with Rpt. ID %1⇒
 for Process %2 to post from directory %3", &RPTINSTANCE, &RPTINFO.ProcessName,⇒
 &RPTINFO.SourceReportPath);
   Else
      MessageBox(0, "", 63, 122, "Not successful for process request for Process⇒
 %1 to post from directory %2", &RPTINFO.ProcessName, &RPTINFO.SourceReportPath);
```

```
            End-If;
```

CHAPTER 23

ProcessRequest Class

This chapter provides an overview of ProcessRequest class and discusses the following topics:

- Data type of a ProcessRequest object.
- Scope of a ProcessRequest object.
- Options for items in jobs and jobsets.
- Values for output type and format .
- Alternative option to specify email or web attributes.
- ProcessRequest class built-in function.
- ProcessRequest class methods.
- ProcessRequest class properties.
- ProcessRequest examples.

Understanding ProcessRequest Class

The ProcessRequest class provides properties and a method for scheduling a pre-defined process or job. You must define the process or the job (using Process Scheduler Manager) *before* you can schedule it using the ProcessRequest class.

The properties of this class contain the same values as those required by Process Scheduler Manager for scheduling a process or job. Values you provide for these properties may override the equivalent values set in Process Scheduler Manager, depending on the override settings you make in PeopleSoft Process Scheduler pages.

Starting with PeopleSoft 8.4, PeopleSoft supports only the ProcessRequest class to schedule a process or a job in PeopleCode. The ScheduleProcess PeopleCode function is no longer supported. Any existing PeopleCode functions containing this deprecated function must be modified to use the ProcessRequest class to schedule a request.

This chapter assumes that the reader has working knowledge of PeopleSoft Process Scheduler.

See Also

PeopleSoft Process Scheduler

Data Type of a ProcessRequest Object

A ProcessRequest object is declared using the ProcessRequest data type. For example:

```
Local ProcessRequest &RQST;
```

Scope of a ProcessRequest Object

A ProcessRequest object can be instantiated only from PeopleCode. However, if you need to access the Process Schedule Manager from outside of PeopleCode, for example, in a Visual Basic program, you can use the Process Schedule Manager API.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Component Interface PeopleCode, record field PeopleCode, and so on.

See Also

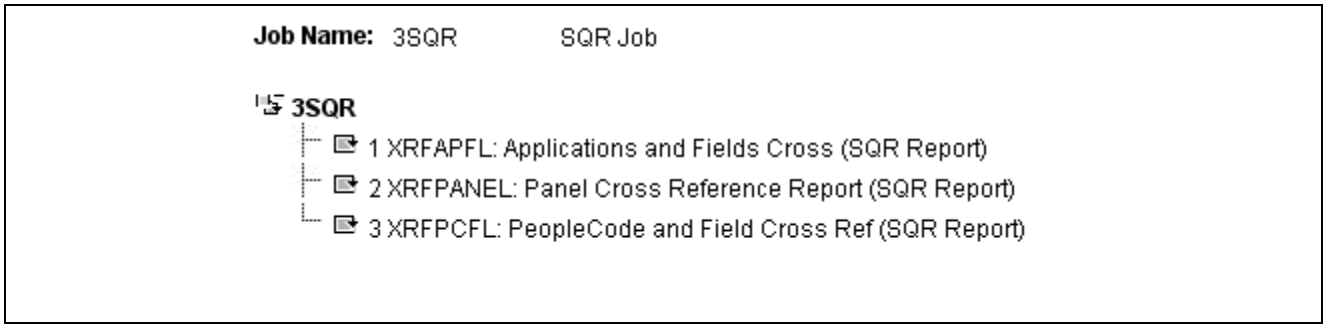
Options for Items In Jobs and Jobsets

A key feature of PeopleSoft Process Scheduler is the option to create a Job Definition consisting of other jobs. A job containing a job is called a *jobset*. Process Scheduler Server schedules the items within these jobs according to how the jobset is defined through the Process Scheduler Manager.

The ProcessRequest class requires the following properties to handle jobsets:

- JobName: indicates the name of the job that an item belongs to.
- PrcsItemLevel: indicates a job item's process item level within a jobset.
- JobSeqNo: indicates a job item's job sequence number within the Process Item Level.

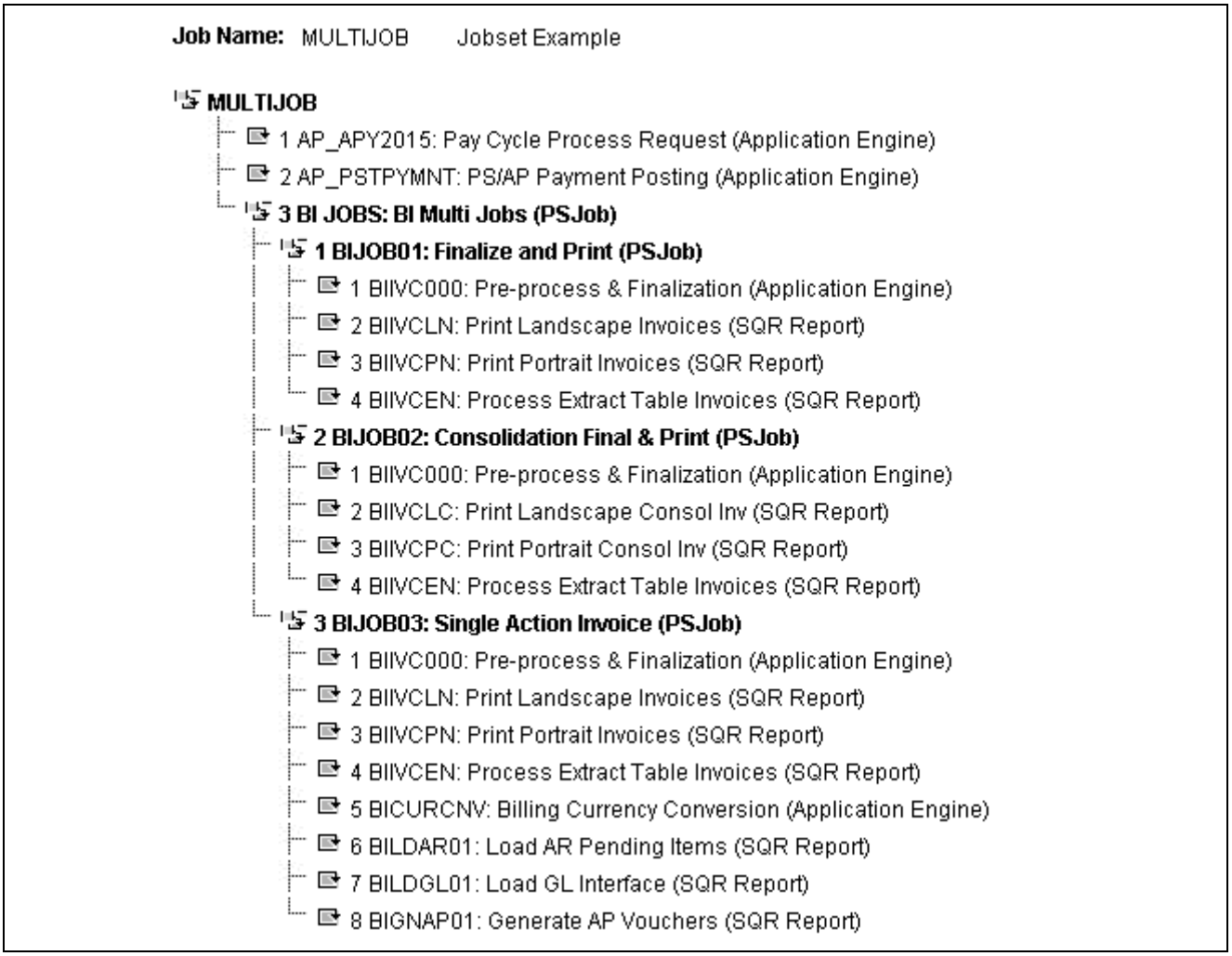
The following is a simple example of a job containing only a single process:



Sample job with single process

Process	PrcsItemLevel	JobName	JobSeqNo
3SQR	0		0
XRFAPFL	1	3SQR	1
XRFPANEL	2	3SQR	2
XRFPCFL	3	3SQR	3

The following is an example of a jobset containing multiple jobs:



Sample jobset containing multiple jobs

Process	PrcsItemLevel	JobName	JobSeqNo
MULTIJOB	0		0
AP_APY2015	1	MULTIJOB	1
AP_PSTPYMNT	1	MULTIJOB	2
BI JOBS	1	MULTIJOB	3
BIJOB01	2	BI JOBS	1
BIIVC000	3	BIJOB01	1
BIIVCLN	3	BIJOB01	2
BIIVCPN	3	BIJOB01	3
BIIVCEN	3	BIJOB01	4
BIJOB02	2	MULTIJOB	2
BIIVC000	3	BIJOB02	1
BIIVCLC	3	BIJOB02	2
BIIVCPC	3	BIJOB02	3
BIIVCEN	3	BIJOB02	4
BIJOB03	2	MULTIJOB	3
BIIVC000	3	BIJOB03	1
BIIVCLN	3	BIJOB03	2
BIIVCPN	3	BIJOB03	3
BIIVCEN	3	BIJOB03	4

Process	PrcsItemLevel	JobName	JobSeqNo
BICURCNV	3	BIJOB03	5
BILDAR01	3	BIJOB03	6
BILDGL01	3	BIJOB03	7
BIGNAP01	3	BIJOB03	8

All the class methods that are used to change options for items within a job or jobset, use these attributes as optional parameters.

If a method is used without specifying any of these parameters, the system assumes the changes apply to *all* items within a jobset.

If one or more of these attributes are specified, the method applies the changes to items matching the value of these attributes.

In the case where a JobName is specified, the changes apply to all items within the specified job.

Specific methods used to manipulate options within a job or jobset include:

- AddDistributionOption
- SetEmailOption
- SetItemFolder
- SetOutputOption

When manipulating the options within a job or jobset, you must specify the process type and process name in the CreateProcessRequest function when instantiating a ProcessRequest object. This is required as the class needs to retrieve the job item information from the job definition for a specific job or jobset. The class uses this information to alter items within the job or jobset based on subsequent method calls made prior to scheduling the request. If the process type and process name are not specified when instantiating the ProcessRequest object, all PeopleCode programs trying to use any of the methods return an error message:

```
The method can not be used because the ProcessRequest is not properly initialized
```

In the previous jobset example, the ProcessRequest must be instantiated as follows:

```
Local ProcessRequest &RQST;

&RQST = CreateProcessRequest("PSJob", "MULTIJOB");
```

The following code examples use the previous example MULTIJOB, and show how items can be altered within a job or jobset:

- Change all items in the Jobset to “Web” with reports generated in PDF format:

  ```
  &RQST.SetOutputOption("Web", "PDF", "", "MULTIJOB");
  ```

  Alternatively, this can also be coded as follows:

```
&RQST.SetOutputOption("Web", "PDF", "");
```

- Change the output option for job BIJOB02 to “Email” with reports generated in HTM format:

```
&RQST.SetOutputOption("Email", "HTM", ", "BIJOB02");
```

- Change the output option for BIIVCPN in job BIJOB01 to “File” in LP (LinePrinter) format:

```
&RQST.SetOutputOption("File", "LP", "BIJOB01", 3, 3);
```

- Change all items in level 3 to “Web” in PS (PostScript) format:

```
&RQST.SetOutputOption("Web", "PS", "", "", 3);
```

See Also

Values for Output Type and Format

The values for output type and format are based on the generic process type assigned to a process type definition. The following table provides a cross-reference listing for all delivered process type definitions.

Generic Process Type	Process Type
AppEngine	Application Engine, Optimization Engine
COBOL	COBOL SQL
Crystal	Crystal, Crystal Check
Cube	CubeBuilder, HyperCube Builder
SQR	SQR Process, SQR Report
Winword	Winword
nVision	nVision, nVision-Report, nVision-ReportBook
Other	Essbase

The following table lists all the values for OutDestType delivered with PeopleTools.

However, the values and defaults can be customized through Process Scheduler Manager. Please refer to the PeopleSoft Process Scheduler documentation for additional information on how to accurately determine the values for your system.

Generic ProcessType	Valid OutDestTypes	Defaults
AppEngine	FILE, WEB	WEB
COBOL	NONE	NONE
Crystal	WEB, EMAIL, FILE, PRINTER	WEB
Cube	NONE	NONE
nVision	WEB, EMAIL, FILE, PRINTER, DEFAULT	DEFAULT
SQR	WEB, EMAIL, FILE, PRINTER	WEB
WinWord	NONE	NONE
OTHER	WEB, EMAIL, FILE, PRINTER, NONE	NONE

Similar to the OutDestType, the formats and defaults for a generic process type can be customized through Process Scheduler Manager. The following table lists the values and defaults as delivered.

Generic ProcessType	OutDestType	Valid OutDestFormats	Default
AppEngine	FILE	PDF, XLS, TXT, HTM	TXT
AppEngine	WEB	PDF, XLS, TXT, HTM	TXT
COBOL	NONE	NONE	NONE
Crystal	EMAIL	DOC, HTM, RPT, RTF, TXT, WKS, XLS, PDF	HTM
Crystal	FILE	DOC, HTM, RPT, RTF, TXT, WKS, XLS, PDF	HTM
Crystal	PRINTER	RPT	RPT
Crystal	WEB	DOC, HTM, RPT, RTF, TXT, WKS, XLS, PDF	HTM

Generic ProcessType	OutDestType	Valid OutDestFormats	Default
Cube	NONE	NONE	NONE
nVision	EMAIL	HTM, XLS	XLS
nVision	FILE	HTM, XLS	XLS
nVision	PRINTER	HTM, XLS	XLS
nVision	WEB	HTM, XLS	XLS
nVision	DEFAULT	DEFAULT	DEFAULT
SQR	EMAIL	CSV, HP, HTM, LP, PDF, PS, SPF,OTHER	PDF
SQR	FILE	CSV, HP, HTM, LP, PDF, PS, SPF,OTHER	PDF
SQR	PRINTER	HP, LP, PS, WP	PS
SQR	WEB	CSV, HP, HTM, LP, PDF, PS, SPF,OTHER	WEB
WinWord	NONE	NONE	NONE
OTHER	NONE	NONE	NONE

Alternative Option To Specify Email or Web Attributes

To set the Email or Web attributes in PeopleTools versions prior to release 8.4, your PeopleCode program was similar to the following:

To Set Attributes For Web

The following example shows how to set web attributes in PeopleTools version prior to release 8.4.

```
&RQST.OutDestType = "Web";
&RQST.OutDestFormat = "PDF";
&RQST.OutDest = "User : VP1,Role :Managers";
```

Set Attributes For Email

The following example shows how to set email attributes in PeopleTools version prior to release 8.4.

```
Local string &Subject ;
Local string &Text;

&Subject = "SQR Report:  Cross Reference Listing";
&Text = "This text will be displayed as the text of this email ";
&RQST.OutDestType = "Email";
&RQST.OutDestFormat = "PDF";
&RQST.OutDest = "User : VP1,Role : MANAGERS";
&RQST.EmailSubject = &Subject;
&RQST.EmailText = &Text;
&RQST.EmailAttachLog = False;
```

In PeopleTools 8.4, methods are provided that set the email or web options. The sample code shown previously can be modified as follows:

Set Attribute for Web

Modify the previous example for setting web attributes as follows:

```
&RQST.SetOutputOption("Web", "PDF", "");
&RQST.AddDistributionOption("User", "QEDMO");
&RQST.AddDistributionOption("Role", "MANAGERS");
```

Set Attribute for Email

Modify the previous example for setting email attributes as followings:

```
Local string &Subject ;
Local string &Text;

&Subject = "SQR Report:  Cross Reference Listing";
&Text = "This text will be displayed as the text of this email ";
&RQST.SetOutputOption("Email", "PDF", "");
&RQST.SetEmailOption(&Subject, &Text, False);
&RQST.AddDistributionOption("User", "QEDMO");
&RQST.AddDistributionOption("Role", "MANAGERS");
```

ProcessRequest Class Built-in Functions

Chapter 1, PeopleCode Built-in Functions, CreateProcessRequest , page 104

Chapter 1, PeopleCode Built-in Functions, GetNextProcessInstance , page 261

Chapter 1, PeopleCode Built-in Functions, SetupScheduleDefnItem , page 437

ProcessRequest Class Methods

In this section, we discuss the ProcessRequest class methods. The methods are discussed in alphabetical order.

AddDistributionOption

Syntax

AddDistributionOption(*DistIdType*, *DistId* [, *JobName*] [, *PrcsItemLevel*] [, *JobSeqNo*])

Description

Use the **AddDistributionOption** method to set the distribution options for any job item in the main job. Distribution options enable you to distribute output in different formats (HTML, PDF, Excel, and so on) to other users based on their user ID or role ID.

This function is valid only if the output destination for the request is routed either to Web or Email.

Parameters

Parameter	Description
DistIdType	Specify the distribution ID type as a string. This identifies if the value passed in the DistID is either a user or role. Values for this parameter are: • User • Role
DistID	Specify the distribution ID as a string.
JobName	Specify the name of the job that this item belongs to as a string.
PrcsItemLevel	Specify the job item's process item level within the main job as a number.
JobSeqNo	Specify the job item's job sequence number within the Process Item Level as a number.

Returns

Returns a number: 0 if successful, 1 if system detected an error in the parameter passed.

Example

The following example grants access to the QEDMO user all reports in MULTIJOB from the Web, while users with role of MANAGERS have access to reports created in the BIJOB03 job.

```
&RQST.SetOutputOption("Web", "PDF", "");
&RQST.AddDistributionOption("User", "QEDMO", "MULTIJOB" );
&RQST.AddDistributionOption("Role", "MANAGERS", "BIJOB03");
```

See Also

Chapter 23, ProcessRequest Class , SetEmailOption, page 1287, Chapter 23, ProcessRequest Class , SetOutputOption, page 1289.

PrintJobHTMLRpt

Syntax

PrintJobHTMLRpt()

Description

Use the **PrintJobHTMLRpt** method to generate an HTML report file displaying all items in a job or jobset in a tree as defined in the Job Definition component.

Parameters

None.

Returns

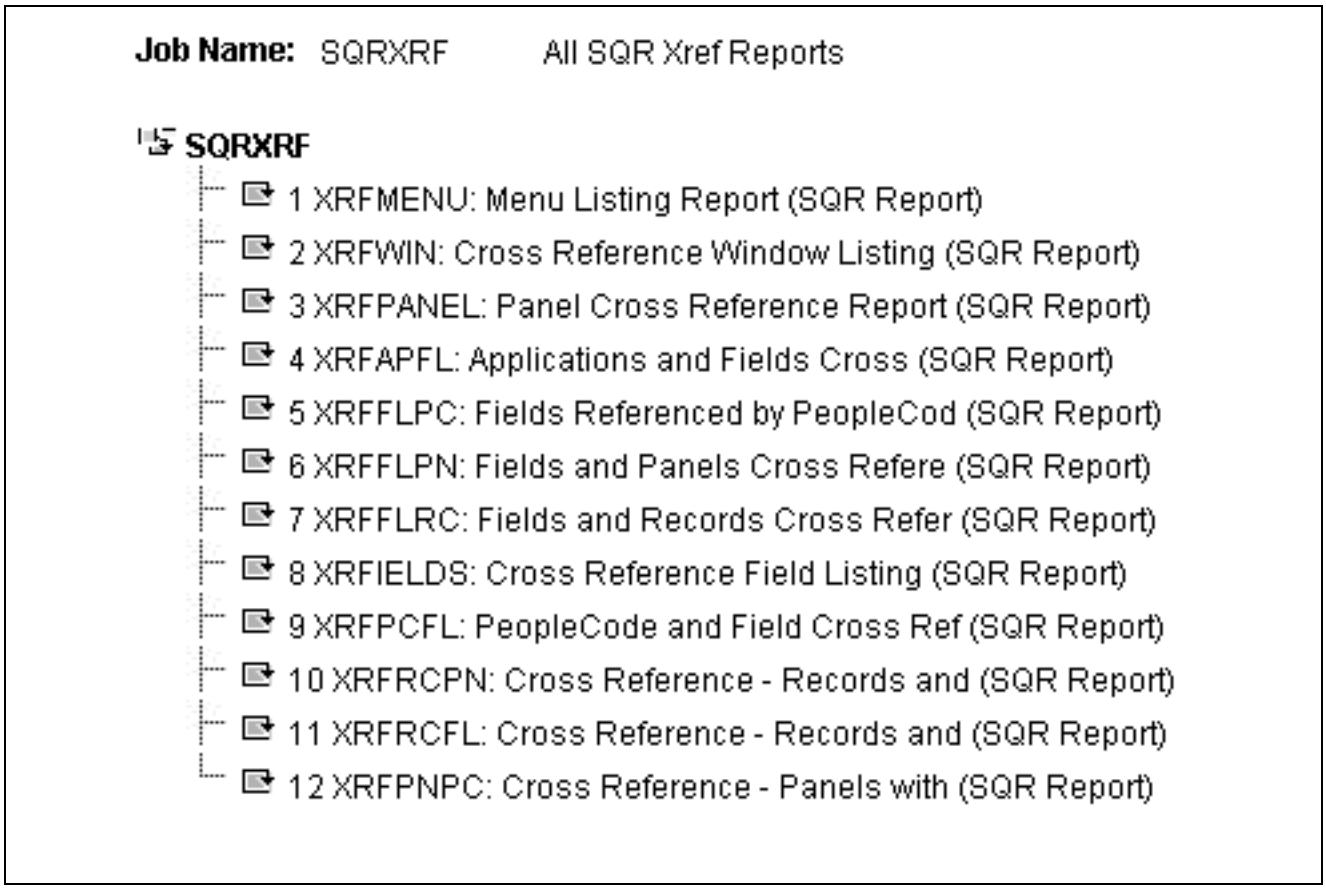
An HTML report as a string

Example

The following code:

```
Local ProcessRequest &JobRQST;
Local string &sHTML;

&JobRQST = CreateProcessRequest("PSJob", "SQRXRF");
&sHTML = &JobRQST.PrintJobHTMLRpt();
```

generates the following HTML file:



Example HTML file

See Also

Chapter 23, ProcessRequest Class , PrintJobRqstRpt, page 1280, Chapter 23, ProcessRequest Class , PrintSchdlHTMLRpt, page 1283.

PrintJobRqstRpt

Syntax

```
PrintJobRqstRpt(ProcessInstance, ItemInstance [, PrintJobTree] [, PrintDistList] [,
PrintNotifyList] [, PrintSystemMessage] [, PrintApplicationMessage] [, PrintParamList])
```

Description

Use the **PrintJobRqstRpt** method to generate an HTML report file displaying the current status of a specific process, job, or jobset.

Parameters

Parameter	Description
ProcessInstance	Specify the ProcessInstance number assigned to the process, job, or jobset
ItemInstance	Specify either to display all items in a job or jobset, or just a specific item. • To display all items, specify 0. • To display a specific item, specify the ProcessInstance number assigned to that item. For a process, set this parameter to 0.
PrintJobTree	Specify whether to have the job tree displayed in the HTML report. This parameter takes a string value: • "1" to display the tree. • "0" to not display the tree. Default value is "1".
PrintDistList	Specify whether you want the list of Users and Roles who will be the recipient of a report generated for a job item displayed in the HTML report. This parameter takes a string value: • "1" to display the list. • "0" to not display the list. Default value is "0".
PrintSystemMessage	Specify whether you want to have the message specified from the Process Definition or Job Definition page displayed in the HTML report. This parameter takes a string value: • "1" to display the system message. • "0" to not display the system message. Default value is "0".
PrintApplicationMessage	Specify whether you want the application messages displayed in the HTML report. These are the application messages that can be viewed from the Message Log subpage of the Process Monitor Detail page. This parameter takes a string value: • "1" to display application messages. • "0" to not display application messages. Default value is "0".
PrintParamList	Specify whether you want the parameter list for a job item displayed in the HTML report. This parameter takes a string value: • "1" to display the parameter list. • "0" to display the parameter list. Default value is "0".

Returns

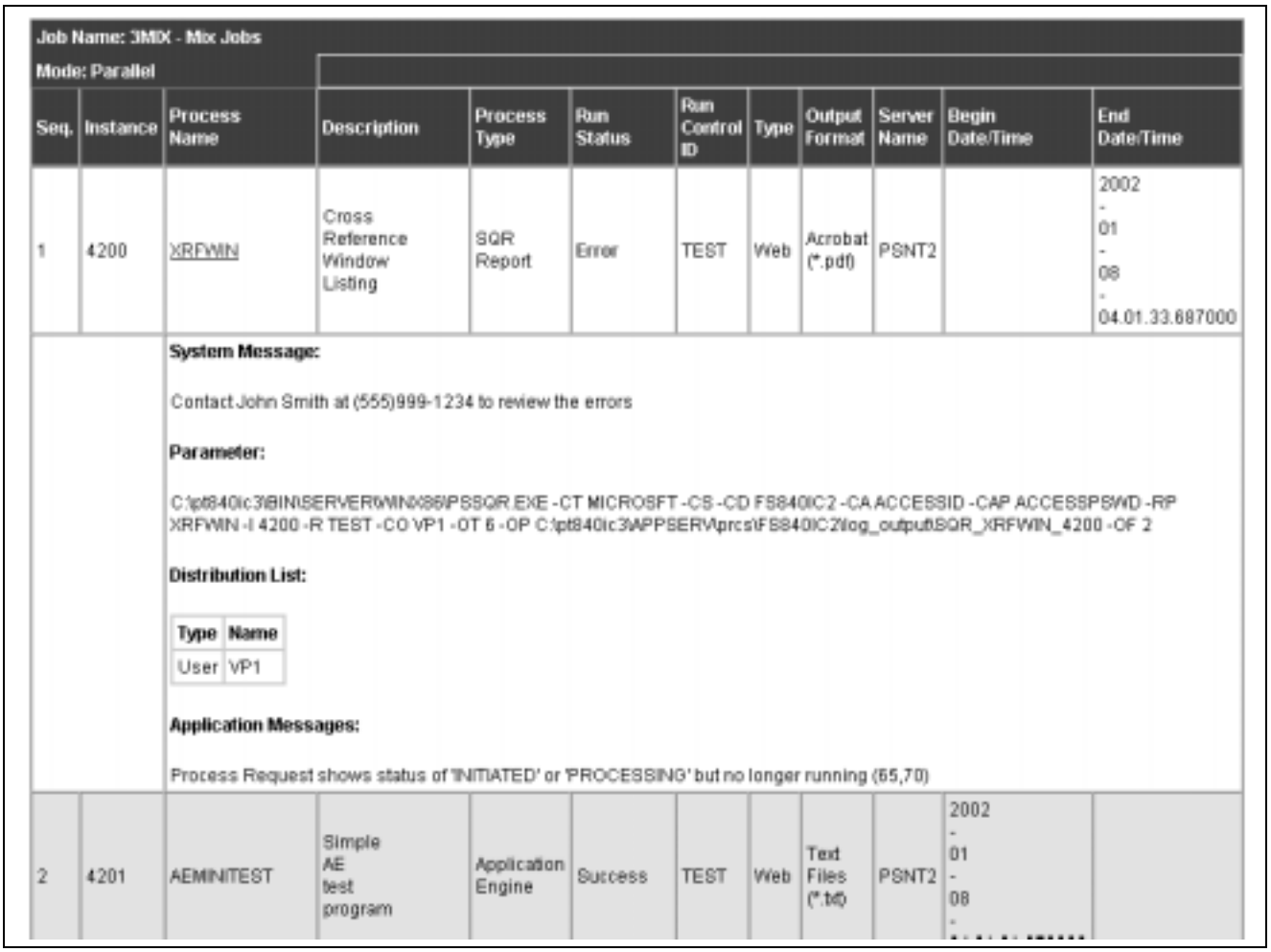
An HTML report as a string.

Example

The following PeopleCode program:

```
Local ProcessRequest &JobRQST;
Local string &sHTML;
Local string &sPRINT_JOBTREE;
Local string &sPRINT_DISTLIST;
Local string &sPRINT_SYSMESSAGE;
Local string &sPRINT_APPLMESSAGE;
Local string &sPRINT_PARAMLIST;

&sPRINT_JOBTREE = "0";
&sPRINT_DISTLIST = "1";
&sPRINT_SYSMESSAGE = "1";
&sPRINT_APPLMESSAGE = "1";
&sPRINT_PARAMLIST = "1";
&JobRQST = CreateProcessRequest();
&sHTML = &JobRQST.PrintJobRqstRpt(PMN_PRCSLIST.PRCSINSTANCE, 0, &sPRINT_JOBTREE,⇒
 &sPRINT_DISTLIST, &sPRINT_SYSMESSAGE, &sPRINT_APPLMESSAGE, &sPRINT_PARAMLIST);
```

Creates the following HTML report:



Job Name: 3MDX - Mix Jobs
Mode: Parallel

Seq.	Instance	Process Name	Description	Process Type	Run Status	Run Control ID	Type	Output Format	Server Name	Begin Date/Time	End Date/Time
1	4200	XRFWIN	Cross Reference Window Listing	SQR Report	Error	TEST	Web	Acrobat (*.pdf)	PSNT2		2002 - 01 - 08 - 04.01.33.687000

System Message:

Contact John Smith at (555)999-1234 to review the errors

Parameter:

C:\pt840ic3\BIN\SERVER\WIN\x86\PSSQR.EXE -CT MICROSFT -CS -CD FS840IC2 -CA ACCESSID -CAP ACCESSPSWD -RP XRFWIN -I 4200 -R TEST -CO VP1 -OT 6 -OP C:\pt840ic3\APPSERV\prcs\FS840IC2\log_output\SQR_XRFWIN_4200 -OF 2

Distribution List:

Type	Name
User	VP1

Application Messages:

Process Request shows status of 'INITIATED' or 'PROCESSING' but no longer running (65,70)

Seq.	Instance	Process Name	Description	Process Type	Run Status	Run Control ID	Type	Output Format	Server Name	Begin Date/Time	End Date/Time
2	4201	AEMINITEST	Simple AE test program	Application Engine	Success	TEST	Web	Text Files (*.txt)	PSNT2	2002 - 01 - 08 -	

Example HTML report

See Also

Chapter 23, ProcessRequest Class , PrintJobHTMLRpt, page 1279, Chapter 23, ProcessRequest Class , PrintSchdlHTMLRpt, page 1283.

PrintSchdlHTMLRpt

Syntax

PrintSchdlHTMLRpt([*PrintJobTree*] [, *PrintDistList*] [, *PrintNotifyList*] [, *PrintMessageList*] [, *PrintParamList*])

Description

Use the **PrintSchdlHTMLRpt** to generate an HTML report file displaying all items in a job or jobset as defined in the Scheduled Jobset Definition component.

Parameters

Parameter	Description
PrintJobTree	Specify whether to have the job tree displayed in the HTML report. This parameter takes a string value: • "1" to display the tree. • "0" to not display the tree. Default value is "1".
PrintDistList	Specify whether you want the list of Users and Roles who will be the recipient of a report generated for a job item displayed in the HTML report. This parameter takes a string value: • "1" to display the list. • "0" to not display the list. Default value is "0".
PrintNotifyList	Specify whether you want the list of Users and Roles who will be notified for the status of a job item displayed in the HTML report. This parameter takes a string value: • "1" to display the list. • "0" to not display the list. Default value is "0".
PrintMessageList	Specify whether you want the messages that will be emailed upon completion of a job item displayed in the HTML report. This parameter takes a string value: • "1" to display messages. • "0" to not display messages. Default value is "0".
PrintParamList	Specify whether you want the parameter list for a job item displayed in the HTML report. This parameter takes a string value: • "1" to display the parameter list. • "0" to display the parameter list. Default value is "0".

Returns

An HTML report as a string

Example

The following code:

```
Local string &sHTML;
Local string &sPRINT_JOBTREE;
Local string &sPRINT_DISTLIST;
Local string &sPRINT_NOTIFYLIST;
Local string &sPRINT_MESSAGELIST;
Local string &sPRINT_PARAMLIST;
```

```
&sPRINT_JOBTREE = "1";
&sPRINT_DISTLIST = "0";
&sPRINT_NOTIFYLIST = "0";
&sPRINT_MESSAGELIST = "0";
&sPRINT_PARAMLIST = "0";
&JobRQST = SetupScheduleDefnItem("Sample", "MULTIJOB");
&sHTML = &JobRQST.PrintSchdlHTMLRpt(&sPRINT_JOBTREE, &sPRINT_DISTLIST, ⇒
&sPRINT_NOTIFYLIST, &sPRINT_MESSAGELIST, &sPRINT_PARAMLIST);
```

Creates the following HTML report:

Job Name: MULTIJOB - Jobset Example

Mode: Serial

Seq.	Process Name	Description	Process Type	Run Control ID	Type	Output Format	Destination	Server Option	Run Time
1	AP_APY2015	Pay Cycle Process Request	Application Engine	RunCntlTest	Web	Text Files (*.txt)	Distribution List	Any Server	
2	AP_PSTPYMNT	PS/AP Payment Posting	Application Engine	RunCntlTest	Web	Text Files (*.txt)	Distribution List	Any Server	
3	BI JOBS	BI Multi Jobs	PSJob	RunCntlTest	(None)	(None)		Any Server	

Job Name: BI JOBS - BI Multi Jobs

Mode: Serial **Parent Job Name: MULTIJOB**

Seq.	Process Name	Description	Process Type	Run Control ID	Type	Output Format	Destination	Server Option	Run Time
1	BIJOB01	Finalize and Print	PSJob	RunCntlTest	(None)	(None)		Any Server	
2	BIJOB02	Consolidation Final & Print	PSJob	RunCntlTest	(None)	(None)		Any Server	
3	BIJOB03	Single Action Invoice	PSJob	RunCntlTest	(None)	(None)		Any Server	

Example HTML report

See Also

Chapter 23, ProcessRequest Class , PrintJobHTMLRpt, page 1279, Chapter 23, ProcessRequest Class , PrintJobRqstRpt, page 1280.

RunJobSetNow

Syntax

RunJobSetNow()

Description

Use the **RunJobSetNow** to schedule a jobset based on settings defined in the Scheduled Jobset Definition component.

Parameters

None.

Returns

None. To verify that the method executed successfully, check the value of the Status property.

Example

```
/* Create the ProcessRequest Object & Run the Scheduled JobSet Now */
Local ProcessRequest &JobRQST;
Local integer &instanceList;

&JobRQST = SetupScheduleDefnItem("Sample","MULTIJOB");
&JobRQST.RunJobSetNow();
&instanceList = &JobRQST.ProcessInstance;
```

See Also

Chapter 23, ProcessRequest Class , Status, page 1299.

Schedule

Syntax

Schedule()

Description

The **Schedule** method inserts a request in the Process Request table which runs according to the values in the properties of the ProcessRequest object. To successfully schedule a process or job, certain properties are required.

- If you’re scheduling a single process, you must assign values to the following properties:
  - RunControlID
  - ProcessName
  - ProcessType
- If you’re scheduling a job where job item changes are not made, you must assign values to the following properties:
  - RunControlID

- JobName

- If you're scheduling a job where job item changes are made, you must assign values to the RunControlID.

Note. You don't have to assign the JobName in this instance because it's set with the CreateProcessRequest function.

Parameters

None.

Returns

None. To verify that the method executed successfully, check the value of the Status property.

Example

```
&MYRQST.Schedule();
If &MYRQST.Status = 0 then
 /* Schedule succeeded. */
Else
 /* Process (job) not scheduled, do error processing */
End-If;
```

See Also

Chapter 23, ProcessRequest Class , RunControlID, page 1297, Chapter 23, ProcessRequest Class , JobName, page 1292, Chapter 23, ProcessRequest Class , ProcessName, page 1296, Chapter 23, ProcessRequest Class , ProcessType, page 1296, Chapter 23, ProcessRequest Class , Status, page 1299, Chapter 1, PeopleCode Built-in Functions, CreateProcessRequest , page 104.

SetEmailOption

Syntax

SetEmailOption(*EmailSubject*, *EmailText*, *EmailAddress*, *EmailWebReport*, *EmailAttachLog* [, *JobName*] [, *PrcsItemLevel*] [, *JobSeqNo*])

Description

Use the **SetEmailOption** method to set the email options for all job items in the main job.

Parameters

Parameter	Description
EmailSubject	Specify the text used in the subject of the email sent at completion of this job email subject as a string. You can specify a Null ("") for this parameter.
EmailText	Specify the email text sent at completion of this job as a string. You can specify a Null ("") for this parameter.
EmailAddress	Specify the email address of the person you want an email sent to at the completion of this job. To send more than one email address, separate the addresses with a semicolon.
EmailWebReport	Specify whether to attach the web report to the email sent at the completion of this job. This parameter takes a Boolean value: True, attach the web report, False, don't attach the web report. This parameter can be set only to True when the **OutDestType** property for the request is "Web".
EmailAttachLog	Specify whether to attach the log file to the email sent at the completion of this job. This parameter takes a Boolean value: True, attach the log file, False, don't attach the log file.
JobName	Specify the name of the job that this item belongs to as a string.
PrcsItemLevel	Specify the job item's process item level within the main job as a number.
JobSeqNo	Specify the job item's job sequence number within the Process Item Level as a number.

Returns

None.

Example

```
&RQST.SetEmailOption("Crystal Report XRFWIN", "Here is XRFWIN", "U:QEDMO; ⇒
R:Employee", False, True, "MULTIJOB", 1, 2);
```

See Also

Chapter 23, ProcessRequest Class , SetItemFolder, page 1288, Chapter 23, ProcessRequest Class , SetOutputOption, page 1289, Chapter 23, ProcessRequest Class , AddDistributionOption, page 1278.

SetItemFolder

Syntax

SetItemFolder(*PortalFolder* [, *JobName*] [, *PrcsItemLevel*] [, *JobSeqNo*])

Description

Use the **SetItemFolder** method to associate a report folder with any job items jobset.

The folder that you specify must have already been created using the System Settings component in Process Scheduler Manager.

Parameters

Parameter	Description
PortalFolder	Specify the name of the report folder to use for the job item as a string. The folder that you specify must have already been created using Report Manager.
JobName	Specify the name of the job that this item belongs to as a string.
PrcsItemLevel	Specify the job item's process item level within the main job as a number.
JobSeqNo	Specify the job item's job sequence number within the Process Item Level as a number.

Returns

None.

Example

```
&RQST.SetItemFolder("GENERAL", "MULTIJOB", 1, 2);
```

See Also

Chapter 23, ProcessRequest Class , AddDistributionOption, page 1278.

SetOutputOption

Syntax

SetOutputOption(*OutputType*, *OutputFormat*, *OutputDest* [, *JobName*] [, *PrcsItemLevel*] [, *JobSeqNo*])

Description

Use the **SetOutputOption** method to modify the output option of one or more job items in the main job.

Note. This method can also be used for a single process.

The job sequence number and job level are optional. They are required only when the same process name is referenced more than once in a job and the user intends to modify the output option of only one item.

If the ProcessRequest object contains an invalid output type for a process, you won't be able to successfully schedule a process or job.

The values for *OutputType*, *OutDestFormat*, and *OutputDestination* are dependent upon each other as well as on other values.

See Chapter 23, ProcessRequest Class , Values for Output Type and Format , page 1274.

Parameters

Parameter	Description
OutputType	Specify the output type as a string.
OutputFormat	Specify the output format as a string.
OutputDestination	Specify the output destination as a string.
JobName	Specify the name of the job that this item belongs to as a string.
PrcsItemLevel	Specify the job item's process item level within the main job as a number.
JobSeqNo	Specify the job item's job sequence number within the Process Item Level as a number.

Returns

None.

Example

```
&RQST.SetOutputOption("Email", "HTM", "", "MULTIJOB", 1, 2);
```

See Also

Chapter 23, ProcessRequest Class , OutDest, page 1293, Chapter 23, ProcessRequest Class , OutDestFormat, page 1294, Chapter 23, ProcessRequest Class , OutDestType, page 1295.

UpdateRunStatus

Syntax

UpdateRunStatus()

Description

Use the **UpdateRunStatus** to change the RunStatus for a specific process, job, or jobset.

Parameters

None.

Returns

None.

Example

```
Local ProcessRequest &RQST;

&RQST = CreateProcessRequest();
&RQST.ProcessInstance = 5;
&RQST.RunStatus = 1; /* Cancel the request */
&RQST.UpdateRunSttus();
```

See Also

Chapter 23, ProcessRequest Class , RunStatus, page 1298.

ProcessRequest Class Properties

In this section, we discuss the ProcessRequest class properties. The properties are discussed in alphabetical order.

EmailAttachLog

Description

This property specifies whether or not a log file is attached to the email sent at completion of this job (or process). This property takes a Boolean value.

This property is read-write.

Example

```
&RQST.EmailAttachLog = False;   /*  Do not attach Log File */
```

EmailSubject

Description

This property specifies the text used in the subject of the email sent at completion of this job (or process). This property takes a string value.

This property is read-write.

Example

```
&RQST.EmailSubject = "SQR Report:  Cross Reference Listing";
```

EmailText

Description

This property specifies the text of the email sent at the completion of this job (or process). This property takes a string value.

This property is read-write.

Example

```
&RQST.EmailText = "This text will be displayed as the text of this email ";
```

You can also use the text from a message in the message catalog for this property.

```
&RQST.EmailText = MsgGetText(65, 117, "Sample text for email with two parameters",⇒
 &MessageParm1, &MessageParm2);
```

EmailWebReport

Description

This property specifies whether an email is sent to recipients of the report after it is posted to the report repository. The URL for the report is included in the email. This option is applicable only when the OutDesType for the request is Web. This property takes a Boolean value: True, the web report should be attached, False otherwise.

This property is read-write.

JobName

Description

This property contains the name of a job that you've defined in PeopleSoft Process Scheduler.

To schedule a job, you must assign values to **JobName** and RunControlID for the Schedule method to succeed.

If you're scheduling a job, you don't need to set the ProcessType property.

This property is read-write.

Example

```
&MYRQST.JobName = "3SQR";
```

See Also

Chapter 23, ProcessRequest Class , RunControlID, page 1297.

LanguageCd

Description

This property enables you to specify a language code for the process or job. If you don't specify a language code, PeopleSoft Process Scheduler first looks in the process run control table to retrieve a language code. If there isn't a value there, PeopleSoft Process Scheduler uses the user's language code specified in the User Definition table.

This property takes a string value.

This property is read-write.

Example

```
&MYRQST.LanguageCd = "ESP" /* Spanish */
```

NotifyTextMsgNum

Description

Use the **NotifyTextMsgNum** property to specify the message that should be displayed to the end user in the Event Notification popup display. This property takes a numeric value.

Note. You must specify both the message set and message number for the message to be displayed to the end user.

This property is read-write.

Example

```
&RQST.NotifyTextMsgSet = 65;
&RQST.NotifyTextMsgNum = 237;
```

See Also

Chapter 23, ProcessRequest Class , NotifyTextMsgSet, page 1293

NotifyTextMsgSet

Description

Use the **NotifyTextMsgSet** method to specify the message set of the text message to be displayed to the end user in the Event Notification popup display. This property takes a numeric value.

Note. You must specify both the message set and message number for the message to be displayed to the end user.

This property is read-write.

Example

```
&RQST.NotifyTextMsgSet = 65;
&RQST.NotifyTextMsgNum = 237;
```

See Also

Chapter 23, ProcessRequest Class , NotifyTextMsgNum, page 1292

OutDest

Description

This property specifies the output destination for the process or job to be scheduled as a string.

Values depend on the setting for the **OutDestType** property:

- If OutDestType is FILE, OutDest must be the name of a directory. If the OutDest property isn't set, it defaults to the setting in the Process profile. If the Process Profile was not updated with a default destination, the Process Scheduler server that picked up the request sets the directory based on the Log/Output Directory setting found in the Process Scheduler Configuration file.
- If OutDestType is PRINTER, OutDest must be the name of a printer. If the OutDest property isn't set, it defaults to the setting in the Process profile. If the Process Profile was not updated with a default printer, the Process Scheduler server that picked up the request sets the printer based on the Default Printer setting found in the Process Scheduler Configuration file.
- If OutDestType is WEB or EMAIL, OutDest should contain the list of User IDs, Role IDs, or email addresses (for email only). If you don't set this property, it's automatically assigned to the person who submitted the request.
- If OutDestType is WEB, PeopleSoft Process Scheduler uses the OutDest property to determine who has access to the output.

- If OutDestType is EMAIL, PeopleSoft Process Scheduler uses the OutDest property to determine who to send the report output to.

  In both these cases, to identify whether it's a User ID or a Role ID, the value has to be preceded by one of the following:

  U: *Username* or **User:** *Username* for a UserID

  R: *Rolename* or **Role:** *Rolename* for a Role ID

  If the ID is *not* preceded by either of these identifiers, Process Scheduler assumes it's an email address.

  Note. Each ID *must* be separated by a semi-colon (;).

  The following are two examples:

```
&RQST.OutDest = "U:PTDMO;R:Employee;robert_smith@peoplesoft.com"

&RQST.OutDest = "User:PS;Role:Employee;sue_line@XYZ.com;simon_gree@peoplesoft.com"
```

- For any other value of OutDestType, this property has no effect.

You can specify directory and printer names using the UNC (Uniform Naming Convention) protocol.

This property is read-write.

Example

```
&MYRQST.OutDest = "C:\TEMP";
```

See Also

Chapter 23, ProcessRequest Class , OutDestType, page 1295.

OutDestFormat

Description

This property specifies the output format for the process or job to be scheduled as a string. Values depend on your settings for the ProcessType and OutDestType properties:

The values for OutDestType, OutDestFormat, and OutDest are dependent upon each other as well as on other values.

See Chapter 23, ProcessRequest Class , Values for Output Type and Format , page 1274.

This property is read-write.

Example

```
&MYRQST.OutDestFormat = "RTF";
```

See Also

Chapter 23, ProcessRequest Class , ProcessType, page 1296, Chapter 23, ProcessRequest Class , OutDestType, page 1295.

OutDestType

Description

This property specifies the type of output for the process or job to be scheduled as a string.

If the ProcessRequest object contains an invalid output type for a process, you won't be able to able to successfully schedule a process or job.

The value specified for this property is used *only* if the Output Destination Type for the defined process or job defined in PeopleSoft Process Scheduler is specified as **Any**. Otherwise any value specified for this property is ignored.

The values for OutDestType, OutDestFormat, and OutDest are dependent upon each other as well as on other values.

See Chapter 23, ProcessRequest Class , Values for Output Type and Format , page 1274.

This property is read-write.

Example

```
&MYRQST.OutDestType = "FILE";
```

See Also

Chapter 23, ProcessRequest Class , ProcessType, page 1296, Chapter 23, ProcessRequest Class , RunControlID, page 1297.

PortalFolder

Description

This property specifies the name of the report folder associated with a job item as a string. If you're specifying a folder, the folder that you specify must have already been created using Report Manager.

This property is read-write.

ProcessInstance

Description

This property is a system-generated identification number. PeopleSoft Process Scheduler assigns a **ProcessInstance** at runtime to each process or job it successfully schedules.

This property is read-write.

Example

```
&MYRQST.Schedule();
If &MYRQST.Status = 0 then
 /* process successfully scheduled */
 &ProcInst = &MYRQST.ProcessInstance;
Else
 /* do error processing */
```

```
End-If;
```

ProcessName

Description

This property specifies the name of a predefined process as a string.

To successfully schedule a process, you must assign values to ProcessName, ProcessType, and RunControlID (that is, for the Schedule method to succeed.)

This property is read-write.

Example

```
&MYRQST.ProcessName = "XRFWIN";
```

See Also

Chapter 23, ProcessRequest Class , ProcessType, page 1296, Chapter 23, ProcessRequest Class , RunControlID, page 1297.

ProcessType

Description

This property specifies the name of a predefined process type as a string.

To successfully schedule a process, you must assign values to ProcessName, ProcessType, and RunControlID (that is, for the Schedule method to succeed.)

The values for ProcessType depend on the types of processes you have defined in your system. There are generic process types that are delivered with your installation of PeopleSoft. These process types may include the following:

- Application Engine
- COBOL
- Crystal
- Cube
- nVision
- SQR
- WinWord
- Other

If you define your own processes, you can use the name of that process with the ProcessType property. For example, suppose you create a custom process named "Custom CBL Programs." You could use this as follows:

```
&MyRqst.ProcessType = "Custom CBL Programs";
```

Note that spaces are included in the string for ProcessType.

This property is read-write.

Example

Note that spaces are included in the string for ProcessType.

```
&MYRQST.ProcessType = "Application Engine";
```

See Also

Chapter 23, ProcessRequest Class , ProcessName, page 1296, Chapter 23, ProcessRequest Class , RunControlID, page 1297.

RunControlID

Description

This property returns a string that serves, along with the UserID, as a key that identifies a predefined group of parameters to be used by a process or a job at runtime.

To successfully schedule a process, you must provide values for RunControlID, ProcessName, and ProcessType.

To successfully schedule a job, you must provide values for RunControlID and JobName.

This property is read-write.

Example

```
&MYRQST.RunControlID = "MYRUNCONTROLID";
or
&MYRQST.RunControlID = PRCSSAMPLEREC.RUN_CNTL_ID;
```

See Also

Chapter 23, ProcessRequest Class , JobName, page 1292, Chapter 23, ProcessRequest Class , ProcessName, page 1296, Chapter 23, ProcessRequest Class , ProcessType, page 1296.

RunDateTime

Description

This property contains a DateTime value that specifies when the scheduled process or job will run.

If you don’t specify a value for this property, and there is no date time set for the pre-defined process or job, the process or job runs as soon as the **Schedule** method is executed.

This property is read-write.

Example

The following example schedules the process or job to run as soon as the Schedule method is executed:

```
&MYRQST.RunDateTime = %Datetime;
```

RunLocation

Description

This property specifies the Process Scheduler Server name the request should be scheduled on. This property takes a string value. Values for RunLocation is a specific server name, such as PSNT.

If no RunLocation is specified, the request is scheduled based on both the default operating system and load balancing options set in the System Settings page.

This property is read-write.

Example

```
&MYRQST.RunLocation = "SERVER";
or
&MYRQST.RunLocation = "PSNT";
```

See Also

.

RunRecurrence

Description

This property specifies the frequency with which a process or job is to be run as a string. The **RunRecurrence** value you use must be the name of a **Recurrence Definition** defined in Process Scheduler Manager to successfully schedule a job or process (that is, for the Schedule method to succeed.)

This property is read-write.

Example

```
&MYRQST.RunRecurrence = "M-F at 5pm";
```

RunStatus

Description

This property specifies the run status of a process request as a number. This property is used with the UpdateRunStatus method to change the status of a request. Values are:

RunStatus	Description
1	Cancel
2	Delete
3	Error

RunStatus	Description
4	Hold
5	Queued
6	Initiated
7	Processing
8	Cancelled
9	Success
10	Not Successful
11	Posted
12	Unable to post
13	Resend
14	Posting
15	Generated
16	Pending

This property is read-write.

Example

```
&MYRQST.RunStatus = 1;
```

See Also

Chapter 23, ProcessRequest Class , UpdateRunStatus, page 1290.

Status

Description

This property returns a number based on the result of the last execution of the Schedule method.

Valid returns are:

- Zero if the method succeeded
- Non-zero otherwise

This property is read-only.

Example

```
&MYRQST.Schedule();
If &MYRQST.Status = 0 then
 /* Schedule succeeded. */
Else
 /* Process (job) not scheduled, do error processing */
End-If;
```

TimeZone

Description

This property contains a timezone value that specifies when the scheduled process or job will run. If no value is used for this property, the server timezone is used. This property takes a string value.

This property is read-write.

Example

```
&MyRqst.TimeZone = "EST";
```

ProcessRequest Class Examples

The following section contains the following examples:

- Scheduling a single process
- Scheduling a job where job item changes are not made
- Scheduling a job where job item changes are made

Scheduling a Single Process

The following example is when you’re scheduling a single process:

```
Local ProcessRequest &RQST;

/* Create the ProcessRequest Object */
&RQST = CreateProcessRequest();

/* Set all the Required Properties */
&RQST.RunControlID = &sRunCntlId;
&RQST.ProcessType = &sProcessType;
```

```
&RQST.ProcessName = &sProcessName;

/* Set any Optional Properties for this Process */
&RQST.RunLocation = &sRunLocation;
&RQST.RunDateTime = &dttmRunDateTime;
&RQST.TimeZone = &sTimeZone;
&RQST.PortalFolder = &sPortalFolder;
&RQST.RunRecurrence = &sRecurrence;
&RQST.OutDestType = &sOutDestType;
&RQST.OutDestFormat = &sOutDestFormat;
&RQST.OutDest = &sOutputDirectory;
&RQST.EmailAttachLog = &bEmailAttachLog;
&RQST.EmailWebReport = &bEmailWebReport;
&RQST.EmailSubject = &sEmailSubject;
&RQST.EmailText = &sEmailText;

/* Schedule the Process */
&RQST.Schedule();
```

Scheduling a Job Where Job Item Changes Are Not Made

The following example is when job item changes aren't made.

```
Local ProcessRequest &RQST;

/* Create the ProcessRequest Object */
&RQST = CreateProcessRequest();

/* Set all the Required Properties */
&RQST.RunControlID = &sRunCntlId;
&RQST.JobName = &sJobName;

/* Set any Optional Main Job Properties */
&RQST.RunLocation = &sRunLocation;
&RQST.RunDateTime = &dttmRunDateTime;
&RQST.TimeZone = &sTimeZone;
&RQST.PortalFolder = &sPortalFolder;
&RQST.RunRecurrence = &sRecurrence;
&RQST.OutDestType = &sOutDestType;
&RQST.OutDestFormat = &sOutDestFormat;
&RQST.OutDest = &sOutputDirectory;
&RQST.EmailAttachLog = &bEmailAttachLog;
&RQST.EmailWebReport = &bEmailWebReport;
&RQST.EmailSubject = &sEmailSubject;
&RQST.EmailText = &sEmailText;

/* Schedule the Job */
&RQST.Schedule();
```

Scheduling a Job Where Job Item Changes Are Made

The following example is when job item changes are made.

```
Local ProcessRequest &RQST;

/* Create the ProcessRequest Object */
&RQST = CreateProcessRequest("PSJob", &sJobName);

/* Set all the Required Properties */
/* Note: the JobName Property has already */
/* been set in the call above, so you don’t */
/* have to set it here */
&RQST.RunControlID = &sRunCntlId;

/* Set any Optional Job Item Properties */
&RQST.SetOutputOption(&sOutputType, &sOutputFormat, &sOutputDest, &sJobName, ⇒
&nPrcsItemLevel, &nJobSeqNo);

&RQST.SetEmailOption(&sEmailSubject, &sEmailText, &sEmailAddress, ⇒
&bEmailWebReport, &bEmailAttachLog, &sJobName, &nPrcsItemLevel, &nJobSeqNo);
&RQST.SetItemFolder(&sPortalFolder, &sJobName, &nPrcsItemLevel, &nJobSeqNo);
&RQST.AddDistributionOption(&sDistIdType, &sDistId, &sJobName, &nPrcsItemLevel, ⇒
&nJobSeqNo);

/* Set any additional Optional Main Job Properties */
&RQST.RunLocation = &sRunLocation;
&RQST.RunDateTime = &dttmRunDateTime;
&RQST.TimeZone = &sTimeZone;
&RQST.PortalFolder = &sPortalFolder;
&RQST.RunRecurrence = &sRecurrence;
&RQST.OutDestType = &sOutDestType;
&RQST.OutDestFormat = &sOutDestFormat;
&RQST.OutDest = &sOutputDirectory;
&RQST.EmailAttachLog = &bEmailAttachLog;
&RQST.EmailWebReport = &bEmailWebReport;
&RQST.EmailSubject = &sEmailSubject;
&RQST.EmailText = &sEmailText;

/* Schedule the Job */
&RQST.Schedule();
```

CHAPTER 24

Query Classes

This chapter provides an overview of Query classes and discusses each class in detail.

Understanding Query Classes

You create queries using PeopleSoft Query Manager to extract the data you need from the PeopleSoft database. You can use the Query classes in PeopleCode to create a new query, or to modify or delete an existing query. You can also use methods in the Query class to execute the query and have the result set returned as either a rowset or have it format and write the result set to a file. You can also use the Query classes to create a SQL statement to be used with the SQL object. In fact, the Query classes expose all of the attributes and methods needed by the PeopleSoft Query Manager and Query Viewer applications.

Creating or deleting a query objectdoes not create or delete query information. You must call the appropriate method for that query object directly to create or delete database information, that is, the **Create** or **Delete** method.

All of the classes, and most of the properties and methods that make up the Query Classes, have a GUI representation in PeopleSoft Query Manager. This document assumes that the reader has working knowledge of PeopleSoft Query Manager.

There aren't any external built-in functions for the Query classes: objects are instantiated from other objects or from a session object.

See Also

Chapter 29, Session Class , page 1559

Collections in the Query Classes

A *collection* is a set of similar things, like a group of already existing queries or QueryRecords. As with everything else in the query classes, collections have a GUI representation in PeopleSoft Query.

For example, when you want to open a query, the search page returns a list of all the available queries. This is equivalent to using the FindQueries session class method to get a Query collection.

The following collections are part of the Query classes:

- QuerySelect collection
- QueryDBRecord collection

- QueryDBRecordField collection
- Query Collection
- QueryRecord collection
- QueryOutputField collection
- QuerySelectedField collection
- QueryCriteria collection
- QueryExpression collection
- QueryPrompt collection

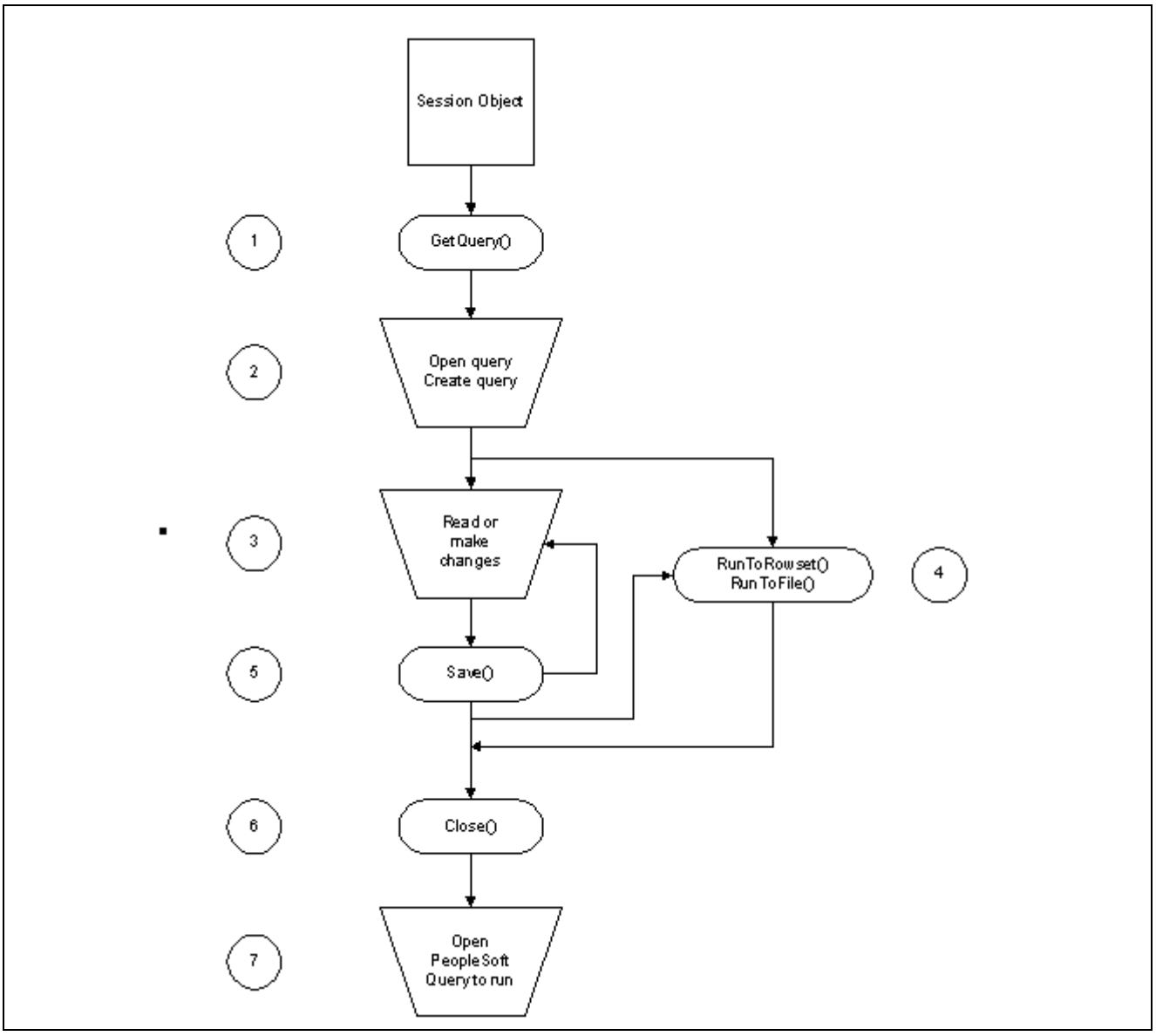
Life-Cycle of a Query

At runtime, there are certain things you want to do with a query, like creating one from scratch, updating the criteria for an existing query, running a query, and so on. The following is an overview of this process, and assumes the most common method to use the Query API. These steps are expanded in other sections.

1. Invoke the GetQuery method on the PeopleSoft session object to get a query.
2. Either open the specific query you want using Open, or create a new query using Create.
3. Read the query statistics, or make changes as appropriate, adding records, field, criteria, and so on.
4. (Optional) Use the RunToRowset method to run the query. You can also use the RunToFile method to execute a scheduled query.

   Note. Be careful whenever you write a PeopleCode program that uses the RunToRowset method because this method could potentially return a large amount of data that could potentially exceed the memory available. For this reason, RunToRowset should be used only when you know that the query being executed returns a reasonable amount of data, or be sure to use the MaxRows parameter to control the maximum amount of data that can be returned.

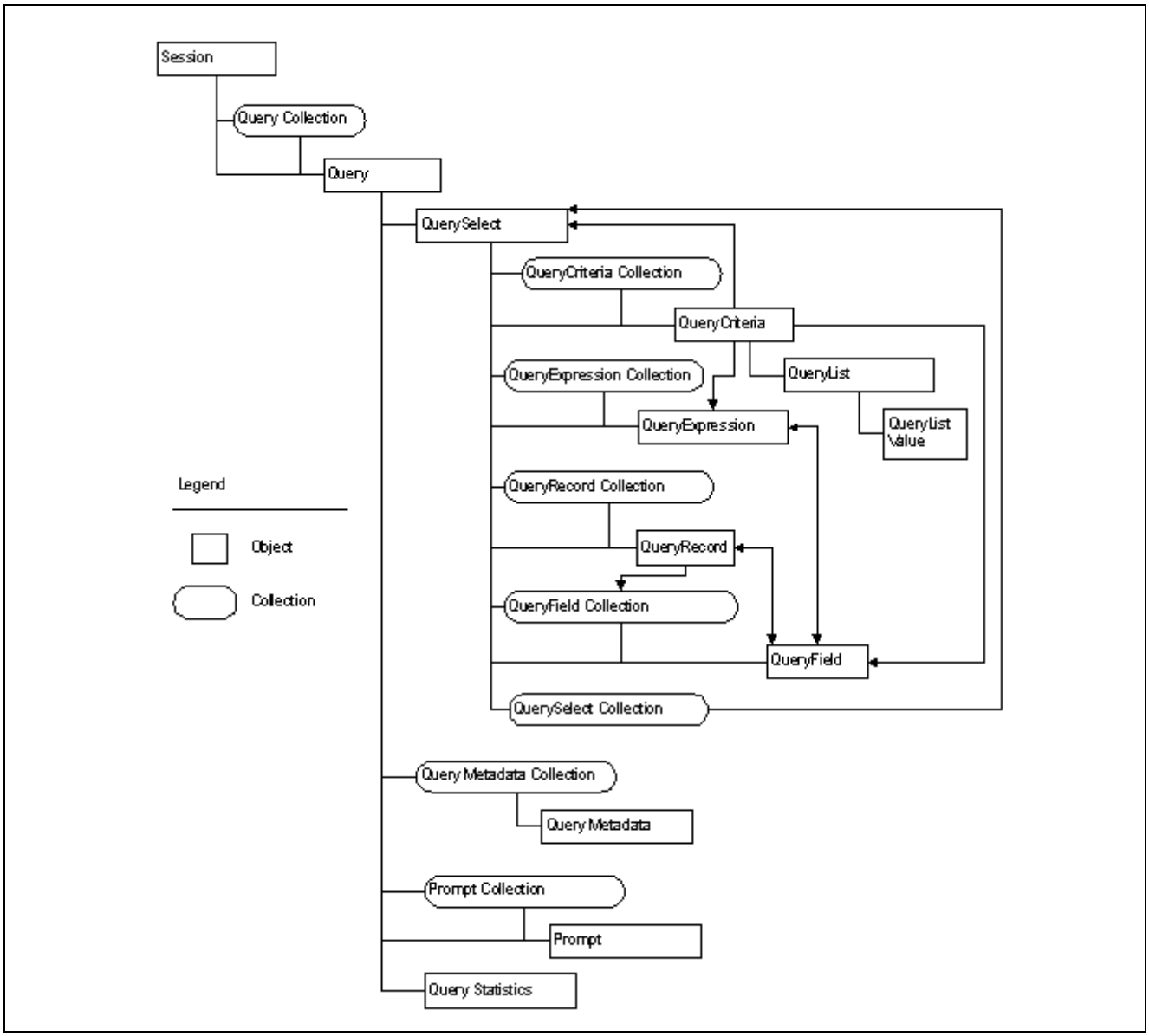
5. Save the changes.
6. Close the query.
7. If you want, you can navigate to PeopleSoft Query Manager, Query Viewer or the Query Designer to run the query.



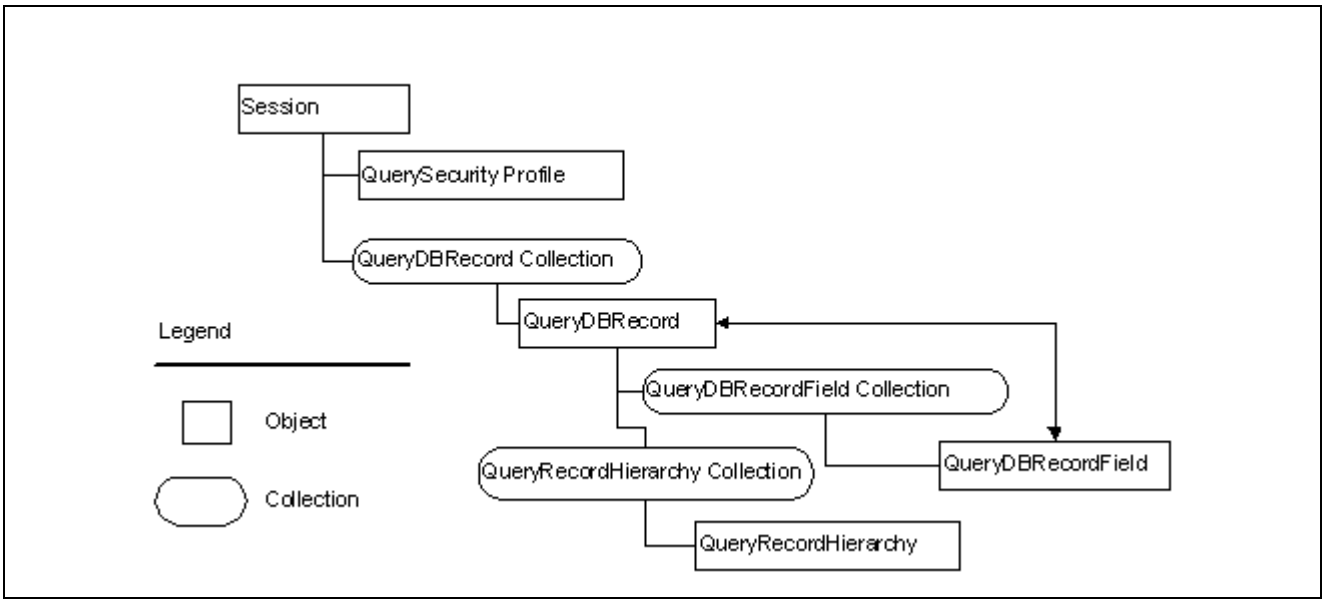
Life cycle of Query API

Query Class Hierarchy

There are many different classes used with the Query API. The following flowcharts illustrate all the different classes and how they're interrelated.



Query API class hierarchy (1 of 2)



Query API class hierarchy (2 of 2)

Query Classes Overview

The Query API is made up of many classes. The following are the primary parts, generally used when updating a query:

Query	The query definition.
QuerySelect	A query SELECT statement. There can be multiple QuerySelect objects for Queries that involve unions or subqueries. Each select (or union or subquery) consists of QueryRecords, QueryOutputFields, QuerySelectedFields, and QueryCriteria.
QueryRecords	The records that are part of the existing QuerySelect definition.
QueryOutputFields	The fields that you've selected to be displayed when the results of the query are run.
QuerySelectedFields	All the fields that make up the QuerySelect Definition. These include the fields selected as output fields, and fields added as part of Query Expressions.
QueryCriteria	The criteria for the query.
QueryExpression	The Query expressions that can contain SQL functions or other SQL fragments.
QueryList	The lists used in the in-list functionality for criteria.
QueryPrompt	The prompts used in criteria.
QueryDBRecords and QueryDBRecordFields	All the records available to be used as QueryRecords, and all the fields available. This list is restricted based on a user's Query Access Security Groups.

Query

A database query. You must get a query before you can access any of the other Query classes. You can then create a new query and save it to the database, or open an existing query and modify it.

QuerySelect

Each query is composed of one or more select statements:

Main Select The instance of the first SELECT statement of the Query is the Main Select. There can only be one Main Select statement in the Query. This instance consists of the QueryOutputFIelds, QueryCriteria, and the QueryExpressions for the Main Select statement.

Union In addition to the Main Select, a query can have one or more unions. The Unions are added by using the AddUnion method.

Sub-Select A subquery used in the criteria of the Main Select or in a Union.

QueryRecords

The records that are part of an existing query definition. In PeopleSoft Query, these are the records listed on the query tab. Each QuerySelect has its own set of QueryRecords.

QueryOutputFields

The fields that you've selected to be part of the query definition. They're called output fields because when you run the query, these are the fields that make up the output columns.

In PeopleSoft Query, these are the fields listed on the Fields tab.

The collection of QueryOutputFields does *not* necessarily include all columns returned in query resultset. This collection only includes columns that have been added to query output collection using the Query Classes or by an end-user designing a query.

PeopleSoft Query can also add additional columns for related language processing and also for translate labels on fields, and so you cannot use the Query Output Collections as a way to discover all of the columns that are returned in the resultset or by executing the SQL generated from a query.

QuerySelectedFields

All the fields that make up the query definition. These include the fields selected as output fields, and fields added as QueryExpressions.

QueryCriteria

The selection criteria for the query. Each QueryCriteria object is made up of the following:

Logical Any criteria objects after the first must include have a Logical value, either AND or OR.

Expression 1	A field or value that you want to base the selection criterion on, that is, Expression1 is the left-hand side of the criterion's comparison.
Operator	A mathematical or other operator used to specify the relationship between Expression 1 and Expression 2.
Expression 2	A field or other value, also called a comparison value, used with Expression 1, that is, Expression2 is the right-hand side of the criterion's comparison.

In PeopleSoft Query, a QueryCriteria is under the Criteria tab.

Expression can be made up of constant values, fields, subqueries, and so on.

See Also

Chapter 24, Query Classes , Working With Criteria and Expressions , page 1309

QueryDBRecords and QueryDBRecordFields

A QueryDBRecord is a record in the database that can be used as a QueryRecord. The list of records is controlled by security: the only records displayed as QueryDBRecords are records accessible by the user.

The QueryDBRecordFields are the fields that make up the QueryDBRecords.

In PeopleSoft Query, the QueryDBRecords are under the Record tab. After you click on the plus sign next to a record, the QueryDBRecordFields are displayed.

Working With Criteria and Expressions

If you run a query after selecting the QueryFields (which executes a SQL statement, such as SELECT EMPLID, DEPTID from PS_QE_EMPLOYEE), the system retrieves *all* the data in those columns; that is, it retrieves the data from every row in the QueryRecord or records because there is no filter limiting the number of rows.

You can select which *rows* of data you want by adding selection criteria to the query.

This document assumes that you know how to use selection criteria. This section discusses working with the QueryCriteria and QueryExpression objects only in the Query API.

See Also

Setting the Type

Before you can add a new expression (either Expression 1 or Expression 2) to your QueryCriteria, you must set the type for the expression.

The following code example adds a new criteria, sets the type for the first expression, adds the first expression to the main select of the query definition, then does the same thing for the second expression.

```
&MyQuerySelect = &MyQuery.QuerySelect;
&MyCriteria = &MyQuerySelect.AddCriteria();
```

```
/* make expression 1 a field */
&MyCriteria.Expr1Type = %Query_ExprField;

/* add the ABSENCE_TYPE field */
&MyCriteria.AddExpr1Field("A", "ABSENCE_TYPE");

/* make it not equal to */
&MyCriteria.Operator = %Query_CondNotEqual;

/* Make expression 2 a constant */
&MyCriteria.Expr2Type = %Query_ExprConstant;
&MyCriteriaExpr = &MyCriteria.AddExpr2Expression1();
&MyCriteriaExpr.Text = "VAC";
```

Considerations Adding New Expressions

When you use any of the QueryCriteria methods to add either a new Expression 1 or Expression 2, you destroy the existing value.

In general, you should use the QueryCriteria methods to add a new Expression 1 or Expression 2 only when you're adding a new criteria to a query.

Operator and Expression 2 Dependencies

Which values are valid for the Expression 2 type (Expr2Type property) depend on the value of the Operator property.

The following table describes which Expr2Type values are valid with which values of Operator.

Operator	Expression 2
equal to not equal to greater than not greater than less than not less than	Constant Field Expression Subquery
Exists not exists	Subquery
Like not like	Constant (with wildcards)

Operator	Expression 2
is null is not null	
in tree not in tree	Tree Option
Eff Date <= Eff Date >= Eff Date < EffDate >	Field Expression Constant Current Date
First Eff Date Last Eff Date	

See Also

Chapter 24, Query Classes , Operator, page 1390

Query Monitor

The Query Classes provide many methods and properties for examining the QueryStatistics, which you can use to report on the average execution time, the last date and time the query was run, and so on.

You can view the statistics for a query before you save it to the database.

See Also

Using Query Metadata

There are two ways of accessing information about a query:

- The Query classes (QueryOutputField, QueryRecord, and so on).
- The Metadata property.

If you use the Metadata property, the information is presented in a different format. It is also read-only. Using this property can be a quick and easy way to access information about a query.

Each Query Metadata object is a name-value pair. *Name* is an indicator of which Query Metadata property you're accessing, and *Value* is the value of that property.

While the value of each Query Metadata property is unique, the name may or may not be. For example, there is only one description (Descr) for each query, so there is only one Query Metadata property with *name* equal to Descr and *value* equal to the description for the query.

However, there may be more than one record for a query. A Query Metadata property exists for each record, with *name* equal to Record and *value* equal to one of the records in the query. The same is true for Input Param, Expression, Field, and Heading.

The following is a simple PeopleCode program to get the Query Metadata for a private query, ADDRESS_TEMP:

```
Local ApiObject &MyQuery, &MyMetacol, &MyMetadata;
Local File &MyFile;

&MyFile = GetFile("Metadata.Txt", "A");

&MyQuery = %Session.GetQuery();

&Rlst = &MyQuery.Open("ADDRESS_TEMP", False, 1);

If &Rlst = 0 Then
   &MyMetacol = &MyQuery.metadata;
   &MyFile.WriteLine("Name              Value");
   &MyFile.WriteLine("----------------------");

   For &i = 1 To &MyMetacol.count
      &MyMetadata = &MyMetacol.item(&i);
      &Name = &MyMetadata.name;
      &Value = &MyMetadata.Value;
      &MyFile.WriteLine(&Name | "              " | &Value);
   End-For;

Else
   WinMessage("Open query not successful");
End-If;
```

Here is the sample output:

```
Name              Value
----------------------
Descr             Temporary address query for testing
LongDescr
Public/Private    QEDMO
LastUpdDttm       2001-11-19-15.06.22.930000
LastUpdOprId      QEDMO
Record            QE_PERS_DATA
Field             QE_EMPLID
Field             QE_NAME
Field             ADDRESS1
Field             ADDRESS2
Field             ADDRESS3
Field             ADDRESS4
```

```
Field           CITY
Field           COUNTY
Field           STATE
Field           QE_ZIP
Field           QE_COUNTRY
Heading         ID
Heading         Name
Heading         Address 1
Heading         Address 2
Heading         Address 3
Heading         Address 4
Heading         City
Heading         County
Heading         St
Heading         QE_ZIP
Heading         QE_COUNTRY
```

Running a Query

With Query API, you have the following options for running a query:

- Use the RunToRowset method

  This method takes a Query Prompt Record as input and returns a PeopleCode rowset containing the query result. You should always use a standalone rowset (that is, one that was created using CreateRowset).

- Use the RunToFile method

  This method takes a Query Prompt Record and a file name as input and writes the query result to the file.

Considerations Using the RunTo Methods

RunToRowset may require a lot of memory for the rowset object, therefore it also takes more processing time. PeopleSoft recommends that RunToRowset not be used for getting large result sets, of the order of 50000 or more.

Both of the RunTo methods can be called to run a query before saving it, that is, it isn't necessary to first save the query.

The last parameter of both RunToRowset and RunToFile is the maximum number of rows to fetch.

- -1 returns all rows regardless of the setting on the security profile.
- 0 returns the maximum number of rows allowed by the security profile.
- >0 is the limit on the number of rows.

See Chapter 24, Query Classes , RunToFile, page 1334.

See Chapter 24, Query Classes , RunToRowset, page 1336.

Specifying User's Language

For scheduled queries, the system uses the language specified in the user's profile. It does not use the language selected during signon. The system also uses the International and Regional settings the user specified using My Personalizations. If no personal setting have been specified, the system uses the default installation international settings.

Note. Most PeopleSoft components can default to international settings from the browser if the user has not set any user specific settings. However, this is not available for scheduled queries or any Process Scheduler processes.

Query Security

Security is critical for your business data. Typically, you don't want everyone in your company to have access to all the data. All the standard security features used with PeopleSoft applications are integrated in the PeopleSoft Query, as well as the Query classes.

In addition, you can use the QuerySecurityProfile Class to view the current user's security profile for PeopleSoft Query. This class doesn't contain any methods, and all the properties are read-only.

See Also

Chapter 24, Query Classes , QuerySecurityProfile Class, page 1410

PeopleTools Security, Understanding PeopleSoft Security

Error Handling With Query Classes

All errors for the Query classes, like the other APIs, are logged in the PSMessages collection, instantiated from a session object. In addition, some methods return error codes. See the individual method description to see if it returns anything.

The query classes log errors "interactively", that is, as they happen. For example, suppose you specified an invalid query name. The error would be logged in the PSMessages collection as soon as you executed the GetQuery method.

When to check for errors is application-specific. However, if you check for errors after every assignment, you may see a performance degradation.

PeopleSoft recommends that you check for invalid prompts after using any of the following methods or properties:

- Save, RunToRowset, or RunToFile Query class methods.
- SQL, RuntimePrompts, or Prompts Query class properties.
- Any of the methods or properties in the QueryPrompts collection.

The easiest way to check for errors is to check the number of messages in the PSMessages collection, using the **Count** property. If the Count is 0, there are no errors.

```
Local ApiObject &MySession;
Local ApiObject &ERRORCOL;
Local ApiObject &Query, &QueryList;

&MySession = %Session;

If &MySession Then
   /* connection is good */

   &QueryList = &MySession.SearchPublicQueries(%Query_ListQuery, %Query_FindName,⇒
 "%", True);

   For &I = 1 to &QueryList.Count
      &Query = &QueryList.Item(&I);

      /* Do processing */

      /* Do error checking */

      &ERRORCOL = &MySession.PSMessages;
      If (&ERRORCOL.Count <> 0) Then
         /* errors occurred – do processing */
      Else
         /* no errors */
      End-If;

   End-For;
Else
   /* do processing for no connection */
End-If;
```

See Also

Chapter 29, Session Class , Error Handling , page 1560

Understanding QueryOutputFields and QuerySelectedFields

If you select a field from a Query Record, it becomes a QueryOutputField, that is, it becomes one of the columns in the SQL output. QuerySelectedFields consist of displayed fields and Query Expression Fields.

QuerySelectedFields and QueryOutputFields have all the same methods and properties, so in this documentation, they're described together under the heading QueryField.

The collection of QueryOutputFields does *not* necessarily include all columns returned in query resultset. This collection only includes columns that have been added to query output collection using the Query Classes or by an end-user designing a query.

PeopleSoft Query can also add additional columns for related language processing and also for translate labels on fields, and so you cannot use the Query Output Collections as a way to discover all of the columns that are returned in the resultset or by executing the SQL generated from a query.

Understanding Having Criteria

You can have Where (simple) criteria or Having criteria in a query. In most cases, you only have a WHERE clause, that is, simple criteria. Having criteria is only used in some special cases of aggregation queries. They represent the Having clause of the SQL statement, similar to the following:

```
Select A.DEPTID, COUNT(A.EMPLID)

FROM PS_QE_EMPLOYEE A

Group by A.DEPTID

HAVING COUNT(A.EMPLID)>5
```

Having Criteria are separated from simple criteria. They're accessed as follows:

- Having criteria are accessed using either the AddHavingCriteria QuerySelect method or HavingCriteria QuerySelect property
- Simple criteria are accessed using either the AddCriteria QuerySelect method or the Criteria QuerySelect property.
- However, the Having criteria and the simple criteria have all the same methods and properties, so in this documentation, they're described together under the heading QueryCriteria.

See Also

Data Type of Query Objects

All query objects, like a query collection, a QueryDBRecord, a QueryExpression, and so on, are declared as type ApiObject. For example,

```
Global ApiObject &MyQuery;

Local ApiObject &MyExpression;
```

Scope of the Query Objects

All query objects can be instantiated from PeopleCode only.

You can use this object anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

You can only instantiate a query object from a session object. You *must* instantiate the session object before you can instantiate a query object or query collection. Use the %Session system variable to connect to the existing session.

```
Local ApiObject &QueryList;
Local ApiObject &MySession;

&MySession = %Session;

If &MySession <> Null Then
   /* connection is good */
   /* Search for public queries of type QUERY, search both the name and⇒
 description for the pattern MyQ%, and do a case insensitive search */
   &QueryList = &MySession.SearchPublicQueries(1, 3, "MyQ%", False);
Else
   /* do error processing */
End-if;
```

Session Class Methods

The following methods are part of the Query API. However, they're used with a Session object.

See Also

Chapter 29, Session Class , page 1559.

FindQueryDBRecords

Syntax

FindQueryDBRecords()

Description

The **FindQueryDBRecords** method returns a reference to a QueryDBRecord collection, filled with zero or more records.

Security applies to the results of this list, that is, you have access to all the records your UserID (permission list) allows you to access.

Parameters

None.

Returns

A reference to a QueryDBRecord collection containing zero or more records.

See Also

Chapter 24, Query Classes , QueryDBRecord Collection, page 1414, Chapter 24, Query Classes , QueryDBRecord Class, page 1417.

FindQueries

Syntax

```
FindQueries()
```

Description

The **FindQueries** method returns a reference to a Query collection, filled with zero or more queries.

FindQueries Considerations

FindQueries returns both public and private queries. It depends on your database whether the private query is returned first or the public query. If you have two queries with the same name, it depends on your database whether the first use of **Item** returns the private or the public query.

Parameters

None.

Returns

A reference to a Query collection containing zero or more queries.

Example

In the following example, all available queries are returned:

```
Local ApiObject &MySession;
Local ApiObject &MyList;

&MySession = %Session
&MyList = &MySession.FindQueries();
```

See Also

Chapter 24, Query Classes , Open, page 1332, Chapter 24, Query Classes , FindQueriesDateRange, page 1318, Chapter 24, Query Classes , Query Collection , page 1324.

FindQueriesDateRange

Syntax

```
FindQueriesDateRange(StartDateString, EndDateString)
```

Description

The **FindQueriesDateRange** method returns a reference to a Query collection, filled with zero or more queries that match the specified date range.

FindQueriesDateRange Considerations

FindQueriesDateRange returns both public and private queries. It depends on your database whether the private query is returned first or the public query. If you have two queries with the same name, it depends on your database whether the first use of **Item** returns the private or the public query.

Parameters

Parameter	Description
StartDateString	Specify the year, month, and day of the beginning date that you want to look for. This parameter takes a string value. You can specify the date either as YYYY-MM-DD *or* YYYY/MM/DD.
EndDateString	Specify the year, month, and day of the end date. This parameter takes a string value. You can specify the date either as YYYY-MM-DD *or*YYYY/MM/DD.

Returns

A reference to a Query collection containing zero or more queries.

Example

```
Local ApiObject &MySession, &QueryList;

&MySession = %Session;

&Start = GetField(VOLUN_ACT_WRK.START_DT_STR).Value;
&End = GetField(VOLUN_ACT_WRK.END_DT_STR).Value;

&QueryList = &MySession.FindQueriesDateRange(&Start, &End);
```

See Also

Chapter 24, Query Classes , FindQueries, page 1318, Chapter 24, Query Classes , Open, page 1332, Chapter 24, Query Classes , Query Collection , page 1324.

GetQuery

Syntax

```
GetQuery()
```

Description

The **GetQuery** method returns an empty query object. After you have an empty query object, you can use it to open an existing query (using the **Open** method) or to create a new query definition (using the **Create** method).

Parameters

None.

Returns

A reference to an empty query object if successful, NULL otherwise.

Example

```
&MyQuery = &MySession.GetQuery();

If &MyQuery.Open("PHONELIST") Then
```

See Also

Chapter 24, Query Classes , FindQueries, page 1318, Chapter 24, Query Classes , FindQueriesDateRange, page 1318, Chapter 24, Query Classes , Open, page 1332, Chapter 24, Query Classes , Create, page 1329.

GetQuerySecurityProfile

Syntax

GetQuerySecurityProfile()

Description

Use **GetQuerySecurityProfile** to return the current user's security profile for PeopleSoft Query. You can then use the QuerySecurityProfile properties to determine if the user can modify queries, the maximum number of rows to fetch for this user, and so on.

Parameters

None.

Returns

A reference to a QuerySecurityProfile if successful, NULL otherwise.

Example

```
&MySecProfile = %Session.GetQuerySecurityProfile();
If &MySecProfile.CanModifyQuery Then
   /* do some processing */
End-If;
```

See Also

Chapter 24, Query Classes , QuerySecurityProfile Class, page 1410.

SearchQueryDBRecords

Syntax

SearchQueryDBRecords(*SearchType*, *Pattern*, *CaseSensitive*)

Description

The **SearchQueryDBRecords** method returns a reference to a QueryDBRecord collection, filled with zero or more records.

Security applies to the results of this list, that is, you have access to all the records your UserID (permission list) allows you to access.

You can use wildcard characters % and _ when searching. % means find all characters, while _ means find a single character. For example, if you wanted to find all queries that started with the letter M, use "M%" for *Pattern*. To find either DATE or DATA, use "DAT_" for *Pattern*.

These characters can be escaped (that is, ignored) using a \. For example, to search for a query that contains the character %, use \% in *Pattern*.

If *Pattern* is an empty string, this method retrieves all queries of the specified type (that is, specifying "" for *Pattern* is the same as specifying "%").

Parameters

Parameter	Description
SearchType	Specify the type of search to be used with the given pattern. You can use either a constant or a number value for this parameter. Values are:

Numeric Value	Constant Value	Description
1	%Query_FindName	Search for records with the name matching the given pattern.
2	%Query_FindDescr	Search for records with the description matching the given pattern.
3	%Query_FindNameDescr	Search for records with either the name or the description matching the given pattern.

Parameter	Description
Pattern	Specify the pattern to be used when searching for records.
CaseSensitive	Specify whether the search is case-sensitive. This parameter takes a Boolean value: True, the search is case-sensitive, False, it isn't.

Returns

A reference to a QueryDBRecord collection containing zero or more records.

Example

```
Local ApiObject &MySession, &DBRecList;

&MySession = %Session;

DBRecList = &MySession.SearchQueryDBRecords(%Query_FindName, "A%", False);
```

See Also

Chapter 24, Query Classes , FindQueryDBRecords, page 1317.

SearchPrivateQueries

Syntax

SearchPrivateQueries(*QueryType, SearchType, Pattern, CaseSensitive*)

Description

The **SearchPrivateQueries** method returns a reference to a Query collection, filled with zero or more Private queries that match the specified *SearchType*, *Pattern*, and *CaseSensitive* choice

Parameters

Parameter	Description
QueryType	Specify the type of query to be searched for. You can specify either a constant or number value for this parameter. Values are:

Numeric Value	Constant Value	Description
1	%Query_ListQuery	Find queries of the type Query.
3	%Query_ListDBAgent	Find queries of the type DB Agent
4	%Query_ListRole	Find queries of the type Role

Parameter	Description
SearchType	Specify the type of search to be used with the given pattern. You can use either a constant or a number value for this parameter. Values are:

Numeric Value	Constant Value	Description
1	%Query_FindName	Search for queries with the name matching the given pattern.
2	%Query_FindDescr	Search for queries with the description matching the given pattern.
3	%Query_FindNameDescr	Search for queries with either the name or the description matching the given pattern.

Parameter	Description
Pattern	Specify the pattern to be used when searching for queries.
CaseSensitive	Specify whether the search is case-sensitive. This parameter takes a Boolean value: True, the search is case-sensitive, False, it isn't.

Returns

A reference to a Query collection containing zero or more queries.

Example

The following example retrieves all private queries of type Query which start with A and do a case-insensitive search, that is, get all queries starting with A or a.

```
Local ApiObject &MySession, &DBRecList;

&MySession = %Session;

DBRecList = &MySession.SearchPrivateQueries(%Query_ListQuery, %Query_FindName,⇒
 "A%", False);
```

See Also

Chapter 24, Query Classes , FindQueries, page 1318, Chapter 24, Query Classes , FindQueriesDateRange, page 1318, Chapter 24, Query Classes , SearchPublicQueries, page 1323.

SearchPublicQueries

Syntax

SearchPublicQueries(*QueryType, SearchType, Pattern, CaseSensitive*)

Description

The **SearchPublicQueries** method returns a reference to a Query collection, filled with zero or more Public queries that match the specified *SearchType*, *Pattern*, and *CaseSensitive* choice.

Parameters

Parameter	Description
QueryType	Specify the type of query to be searched for. You can specify either a constant or number value for this parameter. Values are:

Numeric Value	Constant Value	Description
1	%Query_ListQuery	Find queries of the type Query.
3	%Query_ListDBAgent	Find queries of the type DB Agent
4	%Query_ListRole	Find queries of the type Role

Parameter	Description
SearchType	Specify the type of search to be used with the given pattern. You can use either a constant or a number value for this parameter. Values are:

Numeric Value	Constant Value	Description
1	%Query_FindName	Search for queries with the name matching the given pattern.
2	%Query_FindDescr	Search for queries with the description matching the given pattern.
3	%Query_FindNameDescr	Search for queries with either the name or the description matching the given pattern.

Parameter	Description
Pattern	Specify the pattern to be used when searching for queries.
CaseSensitive	Specify whether the search is case-sensitive. This parameter takes a Boolean value: True, the search is case-sensitive, False, it isn't.

Returns

A reference to a Query collection containing zero or more queries.

Example

The following example retrieves all public queries of type Query that start with A and does a case-insensitive search, that is, get all queries starting with A or a.

```
Local ApiObject &MySession, &DBRecList;

&MySession = %Session;

DBRecList = &MySession.SearchPublicQueries(%Query_ListQuery, %Query_FindName,⇒
 "A%", False);
```

See Also

Chapter 24, Query Classes , FindQueries, page 1318, Chapter 24, Query Classes , FindQueriesDateRange, page 1318, Chapter 24, Query Classes , SearchPrivateQueries, page 1321.

Query Collection

A query collection is returned from the following session methods:

- FindQueries
- FindQueriesDateRange
- SearchPrivateQueries
- SearchPublicQueries

See Also

Chapter 24, Query Classes , FindQueries, page 1318

Chapter 24, Query Classes , FindQueriesDateRange, page 1318

Chapter 24, Query Classes , SearchPublicQueries, page 1323

Chapter 24, Query Classes , SearchPrivateQueries, page 1321

Query Collection Methods

In this section, we discuss each query collection method.

First

Syntax

```
First()
```

Description

The **First** method returns the first Query object in the Query collection.

Parameters

None.

Returns

A reference to a Query object if successful, NULL otherwise.

Example

```
&MyQuery = &MyCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the Query object that exists at the *number* position in the Query collection.

Item Considerations

FindQueries and FindQueriesDateRange return both public and private queries. It depends on your database whether the private query is returned first or the public query. If you have two queries with the same name, it depends on your database whether the first use of **Item** returns the private or the public query.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the Query object that you want returned.

Returns

A reference to a Query object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryColl.Count;
   &MyQuery = &QueryColl.Item(&I);
   /* do processing */
End-For;
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the Query object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing Query within the Query collection. If you specify an invalid name, the object is NULL. The length of this parameter is 30 characters.

Returns

A reference to a Query object if successful, NULL otherwise.

Example

```
&MyQuery = &MyCollection.ItemByName("PHONELIST");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next Query object in the Query collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a Query object if successful, NULL otherwise.

Example

```
&MyQuery = &MyCollection.Next();
```

Query Collection Property

In this section, we discuss the Count query collection property.

Count

Description

This property returns the number of Query objects in the Query Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Query Class

A query object is returned from the following:

- The GetQuery session method.
- The Query Collection methods First, Item, ItemByName, or Next.

See Chapter 24, Query Classes , GetQuery, page 1319.

See Chapter 24, Query Classes , Query Collection , page 1324.

Query Class Methods

In this section, we discuss each Query class method.

AddQuerySelect

Syntax

AddQuerySelect()

Description

Use the **AddQuerySelect** method to add the first Query Select statement into the query definition. This method returns the instance of the newly created QuerySelect.

Parameters

None.

Returns

A reference to a QuerySelect object. If the query already contains the Main Select, it returns NULL.

See Also

Chapter 24, Query Classes , QuerySelect Collection, page 1357.

AddPrompt

Syntax

AddPrompt(*PromptName*)

Description

The **AddPrompt** method adds a prompt with the given name to the query definition. This method returns a new QueryPrompt object that you can then use to specify details about the prompt.

Note. Prompt names are *not* checked for uniqueness. Each new prompt is just added to the list of prompts.

Parameters

Parameter	Description
PromptName	Specify the name of the new prompt with a string. The length of this parameter is 30 characters.

Returns

A reference to a QueryPrompt object.

See Also

Chapter 24, Query Classes , DeletePrompt, page 1330, Chapter 24, Query Classes , Prompts, page 1343, Chapter 24, Query Classes , QueryPrompt Class, page 1428.

Close

Syntax

```
Close()
```

Description

The **Close** method closes the query, freeing the memory associated with that object, and discarding any changes made to the query since the last save. The **Close** method can be used only on an open query, not a closed query. This means you must have opened the query with the Open or Create methods before you can close it. To save any changes, you must use the Save method before using **Close**.

It's very important to close your query when you're finished processing. Canceling out of a page does *not*close a query. You may receive error messages every other time you run your program if you haven't closed your queries.

Parameters

None.

Returns

None.

See Also

Chapter 24, Query Classes , Open, page 1332, Chapter 24, Query Classes , Save, page 1338, Chapter 24, Query Classes , Create, page 1329.

Create

Syntax

Create(*queryname*, *Public*, *Type*, *Description*, *LongDescription*)

Description

The **Create** method creates a new query, based on the parameters passed with the method. The specified query must be a *new* query.

Warning! If you specify the name of a query that already exists, the existing query is overwritten by the new query.

The **Create** method can be used only with a *closed* query, it cannot be used on an open query.

After you create a new query, you don't have to open it with the Open method. The existing query object points to the new query.

Creating a new query doesn't create the query in the database. You must save the query (with the **Save** method) to commit it to the database.

Parameters

Parameter	Description
Queryname	Name of the query to be created. This parameter takes a string value. This parameter takes 30 characters. A query name can contain only alphabetic and numeric characters, as well as underscores.
Public	Specify if the query is public or private. This parameter takes a Boolean value: True if the query is public, False if it's a private query.
Type	Specify the type of query. This parameter takes either a numeric or constant value. Values are:

Numeric Value	Constant Value	Description
1	%Query_Query	Query
3	%Query_View	View
4	%Query_Role	Role
5	%Query_DBAgent	DBAgent

Parameter	Description
Description	Specify a short description for the query. This parameter takes a string value. This parameter takes 30 characters.
LongDescription	Specify a long description for the query. This parameter takes a string value. The length of this parameter depends on your system database limit for LONG fields.

Returns

An integer value: 0 means the query was created successfully.

Example

```
/* Use the existing session */

&MySession = %Session;

&MyQry = &MySession.GetQuery();

/* create a new query : public type-User */
&Rslt = &MyQry.Create("MYQUERY", True, %Query_Query, "My Query", ⇒
"My first Query");

If &Rslt = 0 Then
   /* Query created successfully */
Else
   /* do error processing */
End-If;
```

See Also

Chapter 24, Query Classes , GetQuery, page 1319, Chapter 24, Query Classes , Close, page 1328, Chapter 24, Query Classes , Save, page 1338.

Delete

Syntax

Delete()

Description

The **Delete** method deletes the specified query *from the database*. The **Delete** method can be used only with an *open* query, it cannot be used on a closed query. Before you use the **Delete** method, you must explicitly open the query object to be deleted (with either the Open or Create method.)

Parameters

None.

Returns

An integer value: 0 means the query was deleted successfully.

See Also

Chapter 24, Query Classes , Save, page 1338.

DeletePrompt

Syntax

DeletePrompt(*PromptNumber*)

Description

The **DeletePrompt** method deletes a prompt from a query definition.

Parameters

Parameter	Description
PromptNumber	Specify the numeric location in the query definition of the prompt that you want to delete.

Returns

An integer value: 0 means the query was deleted successfully.

See Also

Chapter 24, Query Classes , AddPrompt, page 1327, Chapter 24, Query Classes , Prompts, page 1343, Chapter 24, Query Classes , QueryPrompt Class, page 1428.

FindExpression

Syntax

```
FindExpression(ExpressionNumber)
```

Description

Use the **FindExpression** method to search the query definition for an expression with the given expression number. Although expressions are associated with the QuerySelect in which they are defined, PeopleSoft Query doesn't qualify expressions by Select number. Therefore, each expression has a unique number across all QuerySelect objects.

Parameters

Parameter	Description
ExpressionNumber	Specify the numeric value of the expression number to be searched for in the query definition.

Returns

A QueryExpression object if successful, NULL otherwise.

See Also

Chapter 24, Query Classes , AddPrompt, page 1327, Chapter 24, Query Classes , Prompts, page 1343, Chapter 24, Query Classes , QueryPrompt Class, page 1428.

FormatResultString

Syntax

```
FormatResultString(&Rowset, Output_Format, StartRow, EndRow)
```

Description

Use the **FormatResultString** method to generate a string containing the content of the input (the rowset) in HTML, XLS, or CSV format. You can specify the range of rows to be in the rowset to be used as input. The rowset object is created using the RunToRowset method.

Parameters

Parameter	Description
&Rowset	Specify an already instantiated and populated rowset object containing the query result. This rowset is created using the RunToRowset method.
Output_Format	Specify the format of the output. You can specify either a numeric or constant value. The values are:

Numeric Value	Constant Value	Description
5	%Query_HTML	The output is in HTML format.
8	%Query_XLS	The output for Excel in HTML format.
14	%Query_TXT	The output for text in CSV format.

Parameter	Description
Start_Row	Specify the first row in the rowset that you want to use for output. This parameter takes a numeric value.
End_Row	Specify the last row in the rowset that you want to use for output. This parameter takes a numeric value.

Returns

A string containing the formatted output.

Example

/* you want to use RowCount, not ActiveRowCount, to get the total number of rows. */

```
&End = &MyRowset.RowCount;

&FormString = &MyQuery.(&MyRowset, %Query_TXT, 1, &End);
```

See Also

Chapter 24, Query Classes , RunToFile, page 1334, Chapter 24, Query Classes , RunToRowset, page 1336.

Open

Syntax

Open(*QueryName*, *Public*, *Update*)

Description

The **Open** method opens the query object specified by the parameters. The **Open** method can be used only with a *closed* query, it cannot be used on an open query. You cannot read or set any properties of a query until after you open it.

Considerations for Opening Different Types of Queries

When you use the Open method, the system tries to open queries according to the type of query. The following is the order used for searching for the query to open:

1. Query (User)
2. View
3. Role
4. DB Agent

All query names are unique. You can't have two queries with the same name, just of different types.

Parameters

Parameter	Description
QueryName	Specify the name of the query to be opened. You must specify an existing query. This parameter takes a string value.
Public	Specify if the query is public or private. This parameter takes a Boolean value: True if the query is public, False if it's a private query.
Update	This parameter is required, but is unused in this release. You must specify a either True or False.

Returns

An integer value: 0 means the query was opened successfully.

See Also

Chapter 24, Query Classes , GetQuery, page 1319, Chapter 24, Query Classes , Close, page 1328, Chapter 24, Query Classes , Save, page 1338.

Rename

Syntax

Rename(*NewQueryName*)

Description

The **Rename** method renames the existing query definition with *NewQueryName*. The **Rename** method can be used only with an *open* query, it cannot be used on a closed query. Before you use the **Rename** method, you must explicitly open the query object to be renamed (with either the Open or Create method.)

Note. The **Rename** takes place immediately. You don't have to save the query for the rename to occur.

Parameters

Parameter	Description
NewQueryName	Specify the new name for the existing query. The maximum length of this parameter is 30 characters.

Returns

An integer value: 0 means the query was renamed successfully.

See Also

Chapter 24, Query Classes , Open, page 1332, Chapter 24, Query Classes , Close, page 1328, Chapter 24, Query Classes , Save, page 1338.

RunToFile

Syntax

```
RunToFile(&PromptRecord, Destination, OutputFormat, MaxRows)
```

Description

Use the **RunToFile** method to execute the Query and return the result to the file specified with *Destination*. The query should be an existing query in the database, or it should have been saved using the **Save** method.

Because a Query may have runtime prompts (that is, criteria defined using Prompt), this method requires those values to be passed in when you execute this method. *PromptRecord* is a PeopleCode record object containing the prompt values as fields in the record. You can use the **PromptRecord** property to obtain this object.

Destination must include the absolute path name of where the file is to be created.

If the specified subdirectory does *not* exist, this method does *not* automatically create them for you, and you receive an error.

If you specify HTML as the output format, the PeopleSoft Query style sheet PSQUERYSTYLEDEF is used for formatting the output.

See Also

PeopleSoft Application Designer, Creating Style Sheet Definitions.

Parameters

Parameter	Description
&PromptRecord	Specify an instance of a PeopleCode record that contains the runtime prompts and values required for running the query.
Destination	Specify the absolute path name of where the files are to be created.
OutputFormat	Specify the format of the data being written to the file. You can use either a constant or a numeric value for this parameter. The values are:

Numeric Value	Constant Value	Description
2	%Query_PDF	The output is in PDF format.
5	%Query_HTML	The output is in HTML format.
8	%Query_XLS	The output for Excel in HTML format.
14	%Query_TXT	The output for txt in CSV format.

Parameter	Description
MaxRows	Specify the maximum number of rows to be fetched. This parameter takes a numeric value. The values are: • -1 returns all rows *regardless* of the setting on the Query security profile (MaxRowsFetched). • 0 returns the maximum number of rows allowed by the Query security profile. • >0 is the limit on the number of rows.

Returns

Returns 0 if successful.

Example

To run a query using the Query API RunToFile method:

1. Open the query.

```
&aRunQry = %Session.GetQuery();

&aRunQry.Open(&sQryName, False, False);
```

2. Obtain the PromptRecord for the query.

```
&aQryPromptRec = &aRunQry.PromptRecord;
```

   This instance of the PromptRecord can be passed to the PeopleCode Prompt function to prompt the user for the runtime values, as follows:

```
&nResult = Prompt(&strQryName | " Prompts", "", &aQryPromptRec);
```

3. Run the query.

   Now that you have the runtime values, the query can be run, as follows:

```
&aRowSet = &aRunQry.RunToFile(&aQryPromptRec, "c:\temp\QueryOutput.html", ⇒
%Query_PDF, 0);
```

4. Access the data of the rowset.

   You can now manipulate the rowset using the data buffer access methods and properties:

```
&aRowSet(&i).GetRecord(1).GetField(&j).Value
```

The following is a complete sample code example:

```
Local Rowset &aRowSet;
   Local Row &aRow;
   Local Record &aQryPromptRec;
   Local Record &aRec;
   Local ApiObject &aRunQry;
   &strHTML = "";
   &aRunQry = %Session.GetQuery();
   If (&aRunQry.Open(&sQryName, False, False) <> 0) Then
```

```
      &strHTML = "Error in opening query";
   Else
      &aQryPromptRec = &aRunQry.PromptRecord;
      &strQryName = &aRunQry.Name;
   If &aQryPromptRec <> Null Then
      &nResult = Prompt(&strQryName | " Prompts", "", &aQryPromptRec);
   End-If;
   If (&aRunQry.RunToFile(&aQryPromptRec, "c:\temp\" | &aRunQry.Name, 3, 0) = 0)⇒
 Then
      &strHTML = "Resultset saved into file successfully.";
   Else
      &strHTML = "Failed to save Resultset into file.";
   End-If;
End-If;
```

See Also

Chapter 24, Query Classes , RunToRowset, page 1336, Chapter 24, Query Classes , PromptRecord, page 1344, Chapter 1, PeopleCode Built-in Functions, Prompt , page 350, *PeopleCode Developer's Guide*, Data Buffer Access.

RunToRowset

Syntax

RunToRowset(*&PromptRecord, MaxRows*)

Description

Use the **RunToRowset** method to execute the Query and return the result to a rowset. The query should be an existing query in the database, or it should have been saved using the **Save** method.

Because a Query may have runtime prompts (that is, criteria defined using Prompt), this method requires those values to be passed in when you execute this method. *PromptRecord* is a PeopleCode record object containing the prompt values as fields in the record. The **PromptRecord** property can be used to obtain this object.

Parameters

Parameter	Description
&PromptRecord	Specify an instance of a PeopleCode record that contains the runtime prompts and values required for running the query.
MaxRows	Specify the maximum number of rows to be fetched. This parameter takes a numeric value. The total number of rows fetched is the minimum value of this parameter's value, and the application server configuration parameter Max Fetch Size (which is specified in KB.) The values are: • -1 returns all rows *regardless* of the setting on the Query security profile (MaxRowsFetched). • 0 returns the maximum number of rows allowed by the Query security profile. • >0 is the limit on the number of rows.

Returns

If successful, the query result is returned as a populated PeopleCode rowset. If the query wasn't successful, the method returns NULL. If unsuccessful, check the PSMessages collection for errors.

Example

To run a query using the Query API RunToRowset method:

1. Open the Query.

```
&aRunQry = %Session.GetQuery();

&aRunQry.Open(&sQryName, False, False);
```

2. Get the PromptRecord for the Query.

```
&aQryPromptRec = &aRunQry.PromptRecord;
```

   This instance of the PromptRecord can be passed to the PeopleCode Prompt function to prompt the user for the runtime values, as follows:

```
&nResult = Prompt(&strQryName | " Prompts", "", &aQryPromptRec);
```

3. Run the query.

   Now that you have the prompt values, you can run the query:

```
&aRowSet = &aRunQry.RunToRowset(&aQryPromptRec, 0);
```

4. Access the data of the rowset.

   You can now manipulate the rowset using the data buffer access methods and properties:

```
&aRowSet(&i).GetRecord(1).GetField(&j).Value
```

The following is a complete sample code example:

```
  Local Rowset &aRowSet;
   Local Row &aRow;
   Local Record &aQryPromptRec;
   Local Record &aRec;
   Local ApiObject &aRunQry;
   &strHTML = "";
   &aRunQry = %Session.GetQuery();
   If (&aRunQry.Open(&sQryName, False, False) <> 0) Then
      &strHTML = "Error in opening query";
   Else
      &aQryPromptRec = &aRunQry.PromptRecord;
      &strQryName = &aRunQry.Name;
   If &aQryPromptRec <> Null Then
      &nResult = Prompt(&strQryName | " Prompts", "", &aQryPromptRec);
      &aRowSet = &aRunQry.RunToRowset(&aQryPromptRec, 0);
         If &aRowSet <> Null Then
               &strHTML = &strHTML | "<table cellpadding='2' cellspacing='0' width=⇒
'90%'>";
```

```
                &aRec = &aRowSet(1).GetRecord(1);
                &QrySelOutputFldCol = &aRunQry.QuerySelect.QueryOutputFields;
                If &QrySelOutputFldCol <> Null Then
                   &strHTML = &strHTML | "<TR><TD>";
                   &strHTML = &strHTML | "<table border='1' cellpadding='2'⇒
 cellspacing='0' width='100%'>";
                   &strHTML = &strHTML | "<TR><TH> </TH>";
                   For &j = 1 To &QrySelOutputFldCol.count
                      &strHTML = &strHTML | "<TH><B>" | &QrySelOutputFldCol.Item⇒
(&j).LongName | "</B></TH>";
                   End-For;
                   &strHTML = &strHTML | "</TR>";
                   &strHTML = &strHTML | "</TD></TR>";
                End-If;
                &strHTML = &strHTML | "<TR><TD>";
                If &aRowSet.RowCount > 0 Then
                   For &i = 1 To &aRowSet.RowCount
                      &aRow = &aRowSet(&i);
                      &strHTML = &strHTML | "<TR>" | "<TD>" | &i | "</TD>";
                      For &j = 1 To &aRow.GetRecord(1).FieldCount
                         &strHTML = &strHTML | "<TD>" | &aRow.GetRecord(1).GetField⇒
(&j).Value | "</TD>";
                      End-For;
                      &strHTML = &strHTML | "</TR>";
                   End-For;
                Else
                   &strHTML = &strHTML | "No records retrieved.";
                End-If;
                &strHTML = &strHTML | "</TD></TR>";
                &strHTML = &strHTML | "</TABLE>";
                &strHTML = &strHTML | "</TABLE>";
             Else
                &strHTML = "Failed to retrieve result set.";
             End-If;
          End-If;
       End-If;
```

See Also

Chapter 24, Query Classes , RunToFile, page 1334, Chapter 24, Query Classes , PromptRecord, page 1344, Chapter 1, PeopleCode Built-in Functions, Prompt , page 350, *PeopleCode Developer's Guide*, Data Buffer Access.

Save

Syntax

Save()

Description

The **Save** method writes any changes to the existing query to the database.

The **Save** method can e used only on an open query, not on a closed query. This means you must have opened the query with the **Open** method before you can save it.

The query object remains open after executing **Save**. You must execute the **Close** method on the object before it is closed and the memory freed.

Note. If you're calling the Query API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Parameters

None.

Returns

An integer value: 0 means the query was saved successfully.

See Also

Chapter 24, Query Classes , Open, page 1332, Chapter 24, Query Classes , Close, page 1328.

Query Class Properties

In the following section, we discuss each Query class property.

Approved

Description

This property returns a string indicating whether a query is approved, unapproved, or modified. This is useful for query administration. The values are:

Value	Description
U	Query is unapproved. This is the default value. The query administrator can prevent execution of unapproved queries.
A	Query has been approved by the query administrator.
M	Query has been modified.

This property is read-write.

ApproveOprId

Description

Note. This property has been deprecated, and remains for backward compatibility only. Use **ApproveUserId** instead.

This property returns a string containing the UserID of the user who approved the query. This can be useful for query administration.

This property is read-write.

See Also

Chapter 24, Query Classes , ApproveUserId, page 1340.

ApproveUserId

Description

This property returns a string containing the UserID of the user who approved the query. This can be useful for query administration.

This property is read-write.

ApproveDtTm

Description

This property returns the date-time stamp when the query was most recently approved. This can be useful for query administration.

This property is read-only.

CreateOprId

Description

Note. This property has been deprecated, and remains for backward compatibility only. Use **CreateUserId** instead.

This property returns a string containing the User Id of the user who created the query. This can be useful for query administration.

This property is read-only.

See Also

Chapter 24, Query Classes , CreateUserId, page 1341.

CreateDtTm

Description

This property returns a string containing the date-time stamp indicating when the query was created. This property is set by the query runtime when a new query is saved. This can be useful for query administration.

This property is read-only.

CreateUserId

Description

This property returns a string containing the User Id of the user who created the query. This can be useful for query administration.

This property is read-only.

Description

Description

This property returns or sets the short description for the query.

The length of this property is 30 characters.

This property is read-write.

ExecAppName

Description

This property specifies the name of the application executing the query. This property isn't required. This means this property is used only if the query will be executed. This name is stored in the query execution log. This property returns a blank value when the query is opened.

When executing a query using the Query API, this property should be set to the Application name that's executing it, so that the Query Monitor can track query execution. Doing this makes this property useful for query administration. If you try to read this property before setting it, you receive a NULL string.

This property should be set before using RunToRowset or RunToFile.

This property is read-write. However, this property is generally used only to set the name, rather than to read it.

ExecLogging

Description

This property indicates whether an execution log should be created when the query is executed. This can be useful for query administration. This property takes a Boolean value: True, create a log, False, don't create one.

The execution log, which is created by setting this property, can be viewed from the Query Monitor. The logging is done in a PeopleSoft Table. It stores, along with other relevant information, the Execution DateTime, Total Execution Time for the query, and Total Fetch Time for the query.

If you try to read this property before setting it, you receive a NULL string.

This property is read-write. However, this property is generally used only to set logging, rather than read.

LastSQLErrorCode

Description

This property returns the last SQL error code returned by the database as a number. The session object contains the error text and any other errors encountered while executing the query.

PeopleSoft recommends checking this value after using RunToRowset or RunToFile.

This property is read-only.

LastUpdDttm

Description

This property returns the most recent date-time when the query was updated as a string.

This property is read-only.

LastUpdOprId

Description

This property returns the User Id of the user who updated the query most recently as a string.

This property is read-only.

LongDescription

Description

This property returns or sets the long description for the query.

The length of this property is 256 characters.

This property is read-write.

Metadata

Description

This property returns a metadata collection.

This property is read-only.

Example

```
&MetadataList = &MyQuery.Metadata;
```

See Also

Chapter 24, Query Classes , Query Metadata Collection, page 1406.

Name

Description

This property returns the name of the query as a string.

The length of this property is 30 characters.

This property is read-only.

PDFFont

Description

This property is used to set the PDF Font number required for generating PDF using RunToFile. This property takes either a numeric or constant value. The values are:

Numeric Value	Constant Value	Description
2	%PDFFont_TraditionalChinese	The output must be written using the Traditional Chinese Font.
3	%PDFFont_SimplifiedChinese	The output must be written using the Simplified Chinese Font.
1	%PDFFont_Japanese	The output must be written using the Japanese Font
4	%PDFFont_Korean	The output must be written using the Korean Font

If you try to read this property before setting it, you receive a NULL string.

This property is read-write.

See Also

Chapter 24, Query Classes , RunToFile, page 1334, Chapter 24, Query Classes , RunToRowset, page 1336.

Prompts

Description

This property returns all the prompts defined for the Query in a QueryPrompt Collection. If you just want the prompts used in the criteria, use the RunTimePrompts property.

This property is read-only.

See Also

Chapter 24, Query Classes , RunTimePrompts, page 1345, Chapter 24, Query Classes , QueryPrompt Collection, page 1426.

PromptRecord

Description

This property returns the runtime prompts of a query as an instance of a PeopleCode record object. This can be used with the Prompt built-in function to prompt for bind values. This record instance can also be used as the first input parameter for the RunToRowset and RunToFile methods.

This property is read-only.

See Also

Chapter 24, Query Classes , RunToFile, page 1334, Chapter 24, Query Classes , RunToRowset, page 1336.

PublicPrivate

Description

This property specifies whether the query is public or private.

You can use either a constant or numeric value for this property. The values are:

Numeric Value	Constant Value	Description
0	%Query_Private	The query is a private query.
1	%Query_Public	The query is a public query.

This property is read-write.

Example

```
If &QryObj.PublicPrivate = %Query_Public Then
   /* code when working with Public Query */
Else
   /* code when working with Private Query */
End-if;
```

QuerySelect

Description

This property returns the Main Select (that is, the first select) of the query definition as a QuerySelect object. For a new query which does not have any selects, it returns NULL.

The Query Records, Query Criteria, QueryOutput fields, and QuerySelected Fields can be obtained using the QuerySelect object only. Hence, after opening a query, this property serves as the starting point for getting the different components of the Select statement.

This property is read-only.

See Also

Chapter 24, Query Classes , QuerySelect Class, page 1346.

QueryStatistics

Description

This property returns statistical information pertaining to the query's execution as a QueryStatistics object.

This property is read-only.

See Also

Chapter 24, Query Classes , QueryStatistics Class, page 1409.

RunTimePrompts

Description

This property returns the prompts used in the Query Definition's criteria in a QueryPrompt Collection. To return all the prompts defined for the query, use the Prompts property.

This is property is read-only.

See Also

Chapter 24, Query Classes , Prompts, page 1343, Chapter 24, Query Classes , QueryPrompt Collection, page 1426.

SQL

Description

This property returns the SQL statement as generated from the query definition as a character string.

Note. The value of the SQL property is never stored as part of the query definition. Instead, it's generated by the system whenever it's needed for one of the RunTo methods or for the SQL property.

In the PeopleSoft Query, this is located under the SQL tab.

This property is read-only.

Considerations Using the SQL Property

This property returns a basic SQL statement. This statement does not include logic for doing related language record transactions or translate descriptions. PeopleSoft does not recommend using this SQL for selecting data. Instead, PeopleSoft recommends using the RunToRowset method for selecting data.

Type

Description

This property specifies the type of query. This parameter takes either a constant or number value. Values are:

Numeric Value	Constant Value	Description
1	%Query_Query	Query of type Query
3	%Query_View	Query of type View

Numeric Value	Constant Value	Description
4	%Query_Role	Role
5	%Query_DBAgent	DBAgent

This property is read-write.

QuerySelect Class

The QuerySelect represents the SELECT statement of the SQL used in the query definition. This can be any of the following:

- The main select statement
- The unions
- The subqueries

Consider the following SQL statement:

```
Select ACCOUNT_NUM from GL_ACCOUNT_TBL_00

Where PRODUCT_ID IN (SELECT DISTINCT PRODUCT_ID FROM ORDER_TBL)

Union

SELECT ACCOUNT_NUM from GL_ACCOUNT_TBL_01
```

In this example there are three select statements. Here, the first statement is the main statement, the second query is the SubQuery, which is the used in the criterion, and the union is the third select statement. There are fields and records associated with each select statement. In the Query API, the first Select is accessible using Query.QuerySelect, which can be obtained as shown:

```
&MainSelect = &Qry.QuerySelect;
```

The Union can be obtained from the main select, as shown:

```
/* There can be more than one union, hence there is collection of selects in the⇒
 main select */

&Union1 = &MainSelect.QuerySelects.Item(1);
```

The subquery is obtained from the main select as shown:

```
/* belongs to first criteria of the select */

&SubQry1 = &MainSelect.Criteria.Item(1). Expr2SubQuery;
```

Therefore, there is one QuerySelect instance for the main select of the Query, one instance per union defined in the query and one instance per sub-query.

In PeopleSoft Query, the first select (that is, the Main Select) is the parent of all the unions and sub-queries. The Main Select is obtained QuerySelect property. The unions and subqueries are obtained from the QuerySelects property of each QuerySelect instance.

A QuerySelect object is returned from the following:

- The QuerySelect Collection methods AddUnion, First, Item, ItemBySelNum or Next
- The QuerySelect Query property

See Chapter 24, Query Classes , QuerySelect Collection, page 1357 and Chapter 24, Query Classes , QuerySelects, page 1356.

QuerySelect Methods

In this section, we discuss each QuerySelect method. The methods are arranged in alphabetical order.

AddAllFields

Syntax

AddAllFields(*QueryRecord*)

Description

The **AddAllFields** method adds all the fields in the specified query record to the query definition, that is, makes them all QueryOutputFields. The query record must already be a part of the query definition.

Parameters

Parameter	Description
QueryRecord	Specify the instance of an existing record in the query from which you want to add all the fields from to the query definition. This instance can be obtained from the QueryRecords collection of the QuerySelect.

Returns

An integer: 0 if successfully added.

See Also

Chapter 24, Query Classes , AddQueryOutputField, page 1350, Chapter 24, Query Classes , AddQueryRecord, page 1350, Chapter 24, Query Classes , AddQuerySelectedField, page 1351.

AddCriteria

Syntax

AddCriteria(*Name*)

Description

The **AddCriteria** method adds new criterion to the query definition. This method returns a reference to a new QueryCriteria object that you can then use to specify details about the criteria.

Note. If you do not specify a unique name for *Name* when adding a criteria, a criteria object referencing the new criteria is returned, not the existing criteria object.

Parameters

Parameter	Description
Name	Specify a string containing the name of the criteria that you want to add. The maximum length of this parameter is 30 characters.

Returns

A reference to a QueryCriteria object.

See Also

Chapter 24, Query Classes , DeleteCriteria, page 1351, Chapter 24, Query Classes , Criteria, page 1354, Chapter 24, Query Classes , QueryCriteria Class, page 1379.

AddExpression

Syntax

```
AddExpression(Name)
```

Description

The **AddExpression** method adds an expression to the query definition. It returns a reference to a new QueryExpression object you can use to specify details about the expression.

Parameters

Parameter	Description
Name	Specify a string containing the name of the expression that you want to add. This maximum length of this parameter is 30 characters.

Returns

A reference to a QueryExpression object.

See Also

Chapter 24, Query Classes , FindExpression, page 1331, Chapter 24, Query Classes , DeleteExpression, page 1352, Chapter 24, Query Classes , Expressions, page 1354, Chapter 24, Query Classes , QueryExpression Class, page 1395.

AddHavingCriteria

Syntax

AddHavingCriteria(*Name*)

Description

The **AddHavingCriteria** method adds new criterion to the query definition, which is intended for use in the Having Clause of the Select statement. This method returns a reference to a new QueryCriteria object that you can then use to specify details about the having criteria.

Note. If you do not specify a unique name for *Name* when adding a criterion, a criteria object referencing the new criteria is returned, not the existing criteria object.

Parameters

Parameter	Description
Name	Specify a string containing the name of the criteria that you want to add. The maximum length of this parameter is 30 characters.

Returns

A reference to a QueryCriteria object.

See Also

Chapter 24, Query Classes , DeleteHavingCriteria, page 1353, Chapter 24, Query Classes , HavingCriteria, page 1355, Chapter 24, Query Classes , QueryCriteria Class, page 1379.

AddQueryExpressionField

Syntax

AddQueryExpressionField(*Name*)

Description

The **AddQueryExpressionField** method adds a field into the query's current Select statement (that is, it's an output field). This field doesn't belong to any database record. It's meant for standalone expressions (that is, the expressions that are explicitly defined in the Query's Expression interface or are added into the left-hand side of criteria).

Parameters

Parameter	Description
Name	Specify the field heading to be used with this expression field, as a string. The maximum length of this parameter is 30 characters.

Returns

A reference to a QueryField object.

See Also

Chapter 24, Query Classes , QueryExpression Class, page 1395, Chapter 24, Query Classes , QueryField Class, page 1368.

AddQueryOutputField

Syntax

```
AddQueryOutputField(QueryRecord, index)
```

Description

The **AddQueryOutputField** method adds a query output field to the query definition. This method returns a reference to a new QueryOutputField object that you can then use to specify attributes for the query definition.

Parameters

Parameter	Description
QueryRecord	Specify the QueryRecord instance that contains the field that you want to include in the query definition. This instance can be obtained from the QueryRecords collection of the QuerySelect.
Index	Specify the numeric position in the QueryRecord of the field that you want to add.

Returns

A reference to a QueryOutputField object.

See Also

Chapter 24, Query Classes , AddAllFields, page 1347, Chapter 24, Query Classes , QueryField Class, page 1368.

AddQueryRecord

Syntax

```
AddQueryRecord(QueryRecordName)
```

Description

The **AddQueryRecord** method adds a query record to the query definition. You must specify a valid record name in the string parameter *QueryRecordName*. This method returns a reference to the record as a QueryRecord.

Parameters

Parameter	Description
QueryRecordName	Specify a string containing the name of a record to be added to the query definition. You must specify a valid record name. The maximum allowed length for this parameter is 15 characters.

Returns

A reference to the record as a QueryRecord object.

See Also
Chapter 24, Query Classes , QueryRecord Class, page 1363.

AddQuerySelectedField

Syntax
AddQuerySelectedField(*QueryRecordName, RecordAlias, FieldName, Heading*)

Description

The **AddQuerySelectedField** method adds a query field to the query definition, as a QuerySelectedField. This method returns a reference to a new QuerySelectedField object that you can then use to specify attributes for the query definition. At the time it's added, it isn't an output field. This can be changed by setting the column number of the field to a value greater than zero.

Parameters

Parameter	Description
QueryRecordName	Specify the name of the query record to which the field that you want to add belongs.
RecordAlias	Specify the alias of the record (such as A, B, C, and so on.) The length of this parameter is 1.
FieldName	Specify the name of the field that's being added. The maximum allowed length for a field name is 18 characters.
Heading	Specify the heading of the field that's being added. The maximum allowed length for a heading is 30 characters.

Returns

A reference to a QuerySelectedField object.

See Also
Chapter 24, Query Classes , AddAllFields, page 1347, Chapter 24, Query Classes , DeleteField, page 1352, Chapter 24, Query Classes , QueryField Class, page 1368.

DeleteCriteria

Syntax
DeleteCriteria(*index*)

Description

The **DeleteCriteria** method deletes the criteria specified by *index* from the query definition.

Parameters

Parameter	Description
Index	Specify the numeric position in the query definition of the criteria that you want to delete.

Returns

An integer value: 0 means the query was deleted successfully.

See Also

Chapter 24, Query Classes , AddCriteria, page 1347, Chapter 24, Query Classes , Criteria, page 1354.

DeleteExpression

Syntax

```
DeleteExpression(index)
```

Description

The **DeleteExpression** method deletes the specified expression from the query definition.

Parameters

Parameter	Description
Index	Specify the numeric position of the expression in the query definition that you want to delete.

Returns

An integer: 0 if successfully deleted.

See Also

Chapter 24, Query Classes , AddExpression, page 1348, Chapter 24, Query Classes , Expressions, page 1354, Chapter 24, Query Classes , QueryExpression Class, page 1395.

DeleteField

Syntax

```
DeleteField(index)
```

Description

The **DeleteField** method deletes the selected field from the query definition. This does *not* delete the field from the QueryRecord, just from the query definition, so it's no longer selected.

Parameters

Parameter	Description
Index	Specify the position number of the field that you want deleted from the query definition.

Returns

An integer value: 0 means the query was deleted successfully.

See Also

Chapter 24, Query Classes , AddAllFields, page 1347, Chapter 24, Query Classes , QueryField Class, page 1368.

DeleteHavingCriteria

Syntax

DeleteHavingCriteria(*index*)

Description

The **DeleteHavingCriteria** method deletes the Having Criteria specified by *index* from the query definition.

Parameters

Parameter	Description
Index	Specify the numeric position in the query definition of the having criteria that you want to delete.

Returns

An integer value: 0 means the criteria was deleted successfully.

See Also

Chapter 24, Query Classes , AddHavingCriteria, page 1349, Chapter 24, Query Classes , HavingCriteria, page 1355.

DeleteRecord

Syntax

DeleteRecord(*index*)

Description

The **DeleteRecord** method deletes the selected record from the query definition. This does *not* delete the record from the database or the QueryDBRecord collection, just from the query definition, so it's no longer selected.

Parameters

Parameter	Description
Index	Specify the position number of the record you want deleted from the query definition.

Returns

An integer value: 0 means the query was deleted successfully.

See Also

Chapter 24, Query Classes , AddQueryRecord, page 1350, Chapter 24, Query Classes , QueryField Class, page 1368.

QuerySelect Properties

In this section, we discuss the QuerySelect properties. The properties are discussed in alphabetical order.

Criteria

Description

This property returns a reference to a QueryCriteria Collection for the simple criteria of the SELECT (that is, the criteria that are used in the where clause of the select statement.)

This property is read-only.

See Also

Chapter 24, Query Classes , QueryCriteria Collection, page 1376.

Distinct

Description

This property specifies if the Select is distinct or not.

This property takes a Boolean value: True if the query is distinct, False if the query isn't distinct.

This property is read-write.

Expressions

Description

This property returns a reference to a QueryExpression Collection.

This property is read-only.

See Also

Chapter 24, Query Classes , QueryExpression Collection, page 1393.

HavingCriteria

Description

This property returns a reference to a QueryCriteria collection populated with Having criteria.

This property is read-only.

See Also

Chapter 24, Query Classes , QueryCriteria Collection, page 1376.

ParentSelectNum

Description

This property returns the select number of the parent select of the current select object. Every QuerySelect is assigned a unique number indicating its place in the hierarchy of select statements. For the Main Select, this number is zero. For unions, this number is 1 (unions aren't allowed in subqueries.)

This property is read-only.

QueryOutputFields

Description

This property returns a reference to a QueryField Collection made up of QueryOutputFields.

The collection of QueryOutputFields does *not* necessarily include all columns returned in query resultset. This collection only includes columns that have been added to query output collection using the Query Classes or by an end-user designing a query.

PeopleSoft Query can also add additional columns for related language processing and also for translate labels on fields, and so you cannot use the Query Output Collections as a way to discover all of the columns that are returned in the resultset or by executing the SQL generated from a query.

This is property is read-only.

See Also

Chapter 24, Query Classes , QueryField Collection, page 1365.

QueryRecords

Description

This property returns a reference to a QueryRecord Collection for the query.

This is property is read-only.

See Also

Chapter 24, Query Classes , QueryRecord Collection, page 1360.

QuerySelectedFields

Description

This property returns a reference to a QueryField Collection made up of QuerySelectedFields.

This is property is read-only.

See Also

Chapter 24, Query Classes , QueryField Collection, page 1365.

QuerySelects

Description

This property returns all the Query Selects that are children of the current Select Statement as a QuerySelect Collection. If the query contains Unions, the QuerySelects of the First Select contains Unions and sub-queries (if any criteria contain sub-queries). For all other selects, this property contains only the sub-queries (because PeopleSoft Query doesn't allow unions for sub-queries).

This is property is read-only.

See Also

Chapter 24, Query Classes , QuerySelect Collection, page 1357.

SelectNum

Description

This property returns the select number of the current select. Every QuerySelect is assigned a unique number indicating its place in the hierarchy of select statements. For the Main Select, this number would be 1.

This property is read-only.

SelectType

Description

This property specifies what type of select statement. You can check for either a numeric or a constant value. The values are:

Numeric Value	Constant Value	Description
1	%Query_SelectMain	This is the Main Select of the query definition.
2	%Query_SubQuery	This is a subquery in the query definition.
3	%Query_Union	This is a Union in the query definition.

This property is read-only.

QuerySelect Collection

A QuerySelect Collection is returned from the QuerySelects property of each QuerySelect instance. It contains the unions and sub-queries, which are children of the current Select statement. PeopleSoft Query allows Unions only with the Main Select, the QuerySelect instance for any subqueries will not have any Unions.

See Chapter 24, Query Classes , QuerySelects, page 1356.

QuerySelect Collection Methods

In this section, we discuss the QuerySelect collection methods. The methods are discussed in alphabetical order.

AddUnion

Syntax

AddUnion()

Description

The **AddUnion** method adds a new QuerySelect object of type Union in the current QuerySelect collection. You can use AddUnion only with the first QuerySelect Collection because PeopleSoft Query doesn't support Unions for subqueries. All unions are considered children of the Main Select.

Parameters

None.

Returns

A reference to a QuerySelect object if successful. If the current QuerySelect Collection does not belong to the first Select statement, it returns NULL. If it fails, it returns NULL.

Example

```
&QryMainSel = &Query.QuerySelect;
&QryMainSelCol = &QryMainSel.QuerySelects;
&QryUnion = &QryMainSelCol.AddUnion();
```

DeleteQuerySelect

Syntax

DeleteQuerySelect(*SelNumber*)

Description

The **DeleteQuerySelect** method deletes the QuerySelect instance represented *SelNumber*.

Parameters

Parameter	Description
SelNumber	Specify the numeric value containing the Select Number of the QuerySelect that you want to delete.

Returns

Returns 0 if successful.

Example

```
&QryMainSel = &Query.QuerySelect;
&QryMainSelCol = &QryMainSel.QuerySelects;
&QryMainSelCol.DeleteQuerySelect(3);
```

First

Syntax

First()

Description

The **First** method returns the first child QuerySelect object in the current QuerySelect's collection.

Parameters

None.

Returns

A reference to a QuerySelect object if successful, NULL otherwise.

Example

```
&MyChildQrySel = &MySelCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the QuerySelect object that exists at the *number* position in the current QuerySelect collection.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the QuerySelect object that you want returned.

Returns

A reference to a QuerySelect object if successful, NULL otherwise.

Example

```
For &I = 1 to &QrySelCol.Count;
   &MyChildQrySel = &QrySelCol.Item(&I);
   /* do processing */
End-For;
```

ItemBySelNum

Syntax

ItemBySelNum(*SelNumber*)

Description

The **ItemBySelNum** method returns the QuerySelect object specified by *the Select Number*. Each Select Statement in a Query Definition is identified by a unique select number.

Parameters

Parameter	Description
SelNumber	Specify the numeric value of the Select Number.

Returns

A reference to a QuerySelect object if successful, NULL otherwise.

Example

```
&MyChildQuerySelect = &MySelCol.ItemBySelNum(3);
```

Next

Syntax

Next()

Description

The **Next** method returns the next QuerySelect object in the current QuerySelect collection. You can use this method only after you have used the **First** method: otherwise the system returns a NULL.

Parameters

None.

Returns

A reference to a QuerySelect object if successful, NULL otherwise.

Example

```
&MyChildQuerySelect = &MySelCol.Next();
```

QuerySelect Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the total number of QuerySelect objects in the current QuerySelect Collection, as a number.

This property is read-only.

Example

```
&COUNTCHILDSELS = &MYSELCOL.Count;
```

QueryRecord Collection

A QueryRecord collection is returned from the QueryRecords QuerySelect class method.

See Chapter 24, Query Classes , QueryRecords, page 1355.

QueryRecord Collection Methods

In this section, we discuss each QueryRecord collection method. The methods are discussed in alphabetical order.

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryRecord object in the QueryRecord collection.

Parameters

None.

Returns

A reference to a QueryRecord object if successful, NULL otherwise.

Example

```
&MyQueryRecord = &MyCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the QueryRecord object that exists at the *number* position in the QueryRecord collection.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the QueryRecord object that you want returned.

Returns

A reference to a QueryRecord object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryRecordColl.Count;
   &MyQueryRecord = &QueryRecordColl.Item(&I);
   /* do processing */
End-For;
```

ItemByAlias

Syntax

ItemByAlias(*Alias*)

Description

The **ItemByAlias** method returns the QueryRecord object specified by *Alias*.

Parameters

Parameter	Description
Alias	Specify the record alias (for example, A, B, C, and so on), used in the SQL statement as a TableName alias.

Returns

A reference to a QueryRecord object if successful, NULL otherwise.

Example

```
&MyQueryRecord = &MyCollection.ItemByAlias("A");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next QueryRecord object in the QueryRecord collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryRecord object if successful, NULL otherwise.

Example

```
&MyQueryRecord = &MyCollection.Next();
```

QueryRecord Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of QueryRecord objects in the QueryRecord Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryRecord Class

A QueryRecord object is returned from the following:

- The QueryRecord Collection methods First, Item, ItemByName, or Next.
- The AddQueryRecord QuerySelect class method.
- The QueryRecord QueryField class method.

See Chapter 24, Query Classes , QueryRecord Collection, page 1360; Chapter 24, Query Classes , AddQueryRecord, page 1350 and Chapter 24, Query Classes , QueryRecord, page 1374.

QueryRecord Class Method

In this section, we discuss the GetField method.

GetField

Syntax

GetField(*Index*)

Description

The **GetField** method returns the QueryField object specified by *index* from the QueryRecord collection.

Parameters

Parameter	Description
Index	Specify the number of the field to be accessed.

Returns

A reference to a QueryField object if successful, NULL otherwise.

Example

```
&QryFieldColl = &QryRec.QueryFields;

For &I = 1 to &QryFieldColl.Count
   &QryField = &QryRec.GetField(&I);
   /* do processing */
End-For;
```

QueryRecord Class Properties

In this section, we discuss each QueryRecord class property. The properties are discussed in alphabetical order.

Description

Description

This property returns the short description of the record as a string.

The length of this property is 30 characters.

This property is read-only.

JoinAlias

Description

This property returns or sets the join record's alias, that is, A, B, C, and so on.

This is applicable in any type of join. The value is a string. The length of this property is 1 character.

This property is read-write.

JoinFieldName

Description

This property returns or sets the join record's field used in the join criteria. This is applicable in lookup-table joins. The value is a string.

The length of this property is 18 characters.

This property is read-write.

JoinType

Description

This property returns or sets the join type. This is applicable in any type of join. You can use either a constant or numeric value.

The values for the join type are:

Numeric Value	Constant Value	Description
1	%Query_JoinNone	Query record isn't joined with any other record.
2	%Query_JoinHierarchy	Query record has a hierarchy join with another record.
3	%Query_JoinRelated	Query record has a lookup table join with another record.
4	%Query_JoinTree	Query record has a tree join with another record.

This property is read-write.

Name

Description

This property returns the name of the record as a string.

The length of this property is 15 characters.

This property is read-only.

QueryFields

Description

This property returns a reference to a QueryField Collection that contains instances of all the fields belonging to the record definition.

This property is read-only.

See Also

Chapter 24, Query Classes , QueryField Collection, page 1365.

RecordAlias

Description

This property returns the alias used for the record in the query (that is, A, B, C, and so on.)

The length of this property is 1 character.

This property is read-write.

QueryField Collection

A QueryField collection is returned from the following:

- The QueryOutputFields of QuerySelect class method.
- The QuerySelectedFields of QuerySelect class method.
- The QueryFields QueryRecord class method.

The collection of QueryOutputFields does *not* necessarily include all columns returned in query resultset. This collection only includes columns that have been added to query output collection using the Query Classes or by an end-user designing a query.

PeopleSoft Query can also add additional columns for related language processing and also for translate labels on fields, and so you cannot use the Query Output Collections as a way to discover all of the columns that are returned in the resultset or by executing the SQL generated from a query.

See Chapter 24, Query Classes , QueryOutputFields, page 1355; Chapter 24, Query Classes , QuerySelectedFields, page 1308 and Chapter 24, Query Classes , QueryFields, page 1365.

QueryField Collection Methods

In this section, we discuss each QueryField method. The methods are discussed in alphabetical order.

First

Syntax

First()

Description

The **First** method returns the first QueryField object in the QueryField collection.

Parameters

None.

Returns

A reference to a QueryField object if successful, NULL otherwise.

Example

```
&MyQueryField = &MyCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the QueryField object that exists at the *number* position in the QueryField collection.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the QueryField object that you want returned.

Returns

A reference to a QueryField object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryFieldColl.Count;
```

```
   &MyQueryField = &QueryFieldColl.Item(&I);
   /* do processing */
End-For;
```

ItemByNameAndAlias

Syntax

ItemByNameAndAlias(*Name, RecordAlias*)

Description

The **ItemByNameAndAlias** method returns the QueryField object with the given *Name* and *Record Alias.*

Parameters

Parameter	Description
Name	Specify the name of an existing QueryField within the QueryField collection. If you specify an invalid name, the object is NULL.
RecordAlias	Alias of the record to which the field belongs

Returns

A reference to a QueryField object if successful, NULL otherwise.

Example

```
&MyQueryField = &MyCollection.ItemByNameAndAlias("PHONELIST", "A");
```

ItemByExpNum

Syntax

ItemByExpNum(*ExpNum*)

Description

The **ItemByExpNum** method returns the QueryField object with the given Expression Number, *ExpNum*.

Parameters

Parameter	Description
ExpNum	Expression Number for the given expression field. Non-expression fields have 0 value for the Expression Number.

Returns

A reference to a QueryField object if successful, NULL otherwise.

Example

```
&QryFld = &QryFldCol.ItemByExpNum(1);
```

Next

Syntax

Next()

Description

The **Next** method returns the next QueryField object in the QueryField collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryField object if successful, NULL otherwise.

Example

```
&MyQueryField = &MyCollection.Next();
```

QueryField Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of QueryField objects in the QueryField Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryField Class

A QueryField object is returned by the following:

- The AddQuerySelectedField of QuerySelect class method.
- The AddQueryOutputField of QuerySelect class method.
- The QueryField Collection methods First, Item, ItemByName or Next.

See Chapter 24, Query Classes , AddQuerySelectedField, page 1351 and Chapter 24, Query Classes , AddQueryOutputField, page 1350.

See Chapter 24, Query Classes , QueryField Collection, page 1365.

QueryField Methods

In this section, we discuss each QueryField method. The methods are discussed in alphabetical order.

AddTranslateExpression

Syntax

AddTranslateExpression(*ExpressionName*)

Description

Use the **AddTranslateExpression** method to add Translate Effective Date Logic expressions for translatable fields.

Parameters

Parameter	Description
ExpressionName	Specify the name of the expression name that you want to add.

Returns

A reference to a QueryExpression object.

See Also

Chapter 24, Query Classes , Expressions, page 1354, Chapter 24, Query Classes , DeleteExpression, page 1352, Chapter 24, Query Classes , AddExpression, page 1348, Chapter 24, Query Classes , AddTranslateField, page 1369, Chapter 24, Query Classes , QueryExpression Class, page 1395.

AddTranslateField

Syntax

AddTranslateField(*FieldName*)

Description

Use the **AddTranslateField** method to add a Translate Effective Date Logic field for a translatable field.

Parameters

Parameter	Description
FieldName	Specify the name of the translate field that you want to add. The maximum allowed length of this parameter is 18 characters.

Returns

A reference to a QueryField object.

See Also

Chapter 24, Query Classes , Expressions, page 1354, Chapter 24, Query Classes , DeleteExpression, page 1352, Chapter 24, Query Classes , AddExpression, page 1348, Chapter 24, Query Classes , AddTranslateExpression, page 1369, Chapter 24, Query Classes , QueryExpression Class, page 1395.

QueryField Class Properties

In this section, we discuss the QueryField class properties. The properties are discussed in alphabetical order.

Aggregate

Description

This property returns or sets the aggregate type for the query field.

This property takes either a constant or numeric value. The values are:

Numeric Value	Constant Value	Description
1	%Query_TtlNone	Query field has no total
2	%Query_TtlSum	Query field is for Sum
3	%Query_TtlCount	Query field is for Count
4	%Query_TtlMin	Query field is for Minimum
5	%Query_TtlMax	Query field is for Maximum
6	%Query_TtlAvg	Query field is for Average

This property is read-write.

ColumnNumber

Description

This property returns or sets the column number for the query field as a number. If the column number is greater than zero, the field is an output field.

This property is read-write.

Description

Description

This property returns the long description of the field as a string. This same value is returned by the Description property of the QueryDBRecordField class.

The length of this property is 30 characters.

This property is read-only.

Decimal

Description

This property returns the decimal positions QueryField as a number. This same value is returned by the Decimal property of the QueryDBRecordField class.

This property is read-only.

ExpNum

Description

This property returns or sets the Expression Number for the field. This value is 0 for non-expression fields. If the field is created for an expression, this value is the same as the expression number of the corresponding expression. This property takes a number value.

This property is read-write.

Flag

Description

This property returns the Use Edit flag for the Query Field. It has the same values as returned by the Flag property of the QueryDBRecordField. This property takes a number value.

This property is read-only.

Format

Description

This property returns the field format for the query field. This property returns a string value. This property is useful for displaying the datatype in a string format.

Values are:

Data Type	Value Returned
Character	“CHAR “
Long Character	"LONG CHAR "
Number	"NUMBER "
Signed number	"SIGNED "

Data Type	Value Returned
Date	"DATE "
Time	"TIME "
Datetime	"DATE/TIME "
Image	"IMAGE "

This property is read-only.

HeadingText

Description

This property returns or sets the heading text for the query field as a string.

The length of this property is 30 characters.

This property is read-write.

HeadingType

Description

This property returns or sets the heading type for the query field. This property takes either a numeric or constant value. The values are:

Numeric Value	Constant Value	Description
1	%Query_HdgNone	Query field has no heading.
2	%Query_HdgText	Query field has a text heading.
3	%Query_HdgRftShort	Query field uses the short RFT heading.
4	%Query_HdgRftLong	Query fields uses the long RFT heading.

This property is read-write.

HeadingUniqueFieldName

Description

This property returns or sets the unique field name for the heading. The default value for this property is the record alias combined with the field name.

The length of this property is 30 characters.

This property is read-write.

Length

Description

This property returns the length of the Query Field. The same value is returned by the Length property of the QueryDBRecordField. This property takes a numeric value.

This property is read-only.

LongName

Description

This property returns the long name of the Query Field. The same value is returned by the Long Name property of the QueryDBRecordField. This property takes a string value.

The length of this property is 30 characters.

This property is read-only.

Name

Description

This property returns the name of the query field as a string.

The length of this property is 18 characters.

This property is read-only.

OrderByDirection

Description

This property returns or sets the order by direction. This property takes a numeric value. The constants are the ASCII codes for space (" ") and "D".

Values are:

Numeric Value	Constant Value	Description
32	Code (" ")	Ascending
68	Code("D")	Descending

This property is read-write.

Example

The following sets the order by direction to be ascending.

```
&QueryField.OrderByDirection = Code(" ");
```

The following sets the order by direction to be descending.

```
&QueryField.OrderByDirection = Code("D");
```

OrderByNumber

Description

Use this property to specify whether a field is used as part of an 'Order By' statement in the SQL. The number value of this property indicates which is the first field in the order by statement, which is the second, and so on.

This property is read-write.

Example

To order a query by one field, you must set the OrderByNumber property for the other QueryFields as well, specifying the primary field as 1, the next field as 2, and so on.

```
&QryFld = &MainQrySel.AddQuerySelectedField("PSRECDEFN", "A", "RECNAME", "Record⇒
 Name");
&QryFld.ColumnNumber = 1;
&QryFld.OrderByNumber = 1;

&QryFld = &MainQrySel.AddQuerySelectedField("PSRECDEFN", "A", "RECDESCR", "Record⇒
 Descr");
&QryFld.ColumnNumber = 2;
&QryFld.OrderByNumber = 2;

&QryFld = &MainQrySel.AddQuerySelectedField("PSRECDEFN", "A", "RELLANGRECNAME",⇒
 "Record Lang Rec");
&QryFld.ColumnNumber = 3;
&QryFld.OrderByNumber = 3;
```

QueryRecord

Description

This property returns a reference to the QueryRecord containing this QueryField. If the field has no QueryRecord (that is, that this field is an expression field), this property returns NULL.

This property is read-only.

Example

```
&QryRcd = &QryFld.QueryRecord;
```

RecordAlias

Description

This property returns the record alias for the Query Field as a string. This value is usually a value like "A", "B", and so on. The same value is returned by the RecordAlias property for QueryRecord.

The length of this property is 1 character.

This property is read-only.

ShortName

Description

This property returns the short name of the Query Field. The same value is returned by the ShortName property for QueryDBRecordField. This property takes a string value.

The length of this property is 15 characters.

This property is read-only.

TranslateEffDtLogic

Description

This property returns or sets the effective date logic for the QueryField.

The values are:

Numeric Value	Constant Value	Description
1	%Query_ExprCurDt	Current Date
2	%Query_ExprField	Field
3	%Query_Expression	Expression

This property is read-write.

TranslateExpression

Description

This property returns a reference to an expression object based on a translate field if the Translate Effective Date option refers to an Expression

This property is read-only.

See Also

Chapter 24, Query Classes , QueryExpression Class, page 1395.

TranslateField

Description

This property returns a reference to a QueryField object for a translate field if the Translate Effective Date Option refers to a field.

This property is read-only.

See Also

Chapter 24, Query Classes , QueryField Class, page 1368.

TranslateOption

Description

This property returns or sets the translate value for the QueryField. This property takes a numeric or constant value. The values are:

Numeric Value	Constant Value	Description
1	%Query_XlatNone	None
2	%Query_XlatShort	Short
3	%Query_XlatLong	Long

This property is read-write.

Type

Description

This property returns the type of the field. You can use either a numeric or constant value. The values are:

Numeric Value	Constant Value	Description
0	%FieldType_Char	Character
1	%FieldType_LongChar	Long Character
2	%FieldType_Number	Number
3	%FieldType_SignedNumber	Signed number
4	%FieldType_Date	Date
5	%FieldType_Time	Time
6	%FieldType_DateTime	Datetime
7	%FieldType_Image	Image

This property is read-only.

QueryCriteria Collection

A QueryCriteria collection is returned from the Criteria QuerySelect class property.

See Chapter 24, Query Classes , Criteria, page 1354.

QueryCriteria Collection Methods

In this section, we discuss the QueryCriteria collection methods. The methods are discussed in alphabetical order.

First

Syntax

First()

Description

The **First** method returns the first QueryCriteria object in the QueryCriteria collection.

Parameters

None.

Returns

A reference to a QueryCriteria object if successful, NULL otherwise.

Example

```
&MyQueryCriteria = &MyCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the QueryCriteria object that exists at the *number* position in the QueryCriteria collection.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the QueryCriteria object that you want returned.

Returns

A reference to a QueryCriteria object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryCriteriaColl.Count;
   &MyQueryCriteria = &QueryCriteriaColl.Item(&I);
   /* do processing */
End-For;
```

ItemByName

Syntax

ItemByName(*CriteriaName*)

Description

The **ItemByName** method returns the QueryCriteria object that exists with the passed CriteriaName in the QueryCriteria collection.

Parameters

Parameter	Description
CriteriaName	Specify the name of the Criteria to be searched. This name is the same as the one used while creating the criteria using QuerySelect.AddCriteria(CriteriaName).

Returns

A reference to a QueryCriteria object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryCriteriaColl.Count;
   &MyQueryCriteria = &QueryCriteriaColl.Item(&I);
   /* do processing */
End-For;
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next QueryCriteria object in the QueryCriteria collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryCriteria object if successful, NULL otherwise.

Example

```
&MyQueryCriteria = &MyCollection.Next();
```

QueryCriteria Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of QueryCriteria objects in the QueryCriteria Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryCriteria Class

A QueryCriteria object is returned from the following:

- The AddCriteria and AddHavingCriteria QuerySelect class method.
- The First, Item, and Next method of the QueryCriteria Collection.

See Chapter 24, Query Classes , AddCriteria, page 1347 and Chapter 24, Query Classes , AddHavingCriteria, page 1349.

See Chapter 24, Query Classes , QueryCriteria Collection, page 1376.

See Chapter 24, Query Classes , Working With Criteria and Expressions , page 1309.

See .

QueryCriteria Class Methods

In this section, we discuss the QueryCriteria class methods. The methods are discussed in alphabetical order.

AddExpr1Expression

Syntax

```
AddExpr1Expression()
```

Description

The **AddExpr1Expression** method returns a reference to a new a QueryExpression object to be used as an expression for Expression 1. You can then use this object to specify details about the expression using the methods and properties of the QueryExpression Class.

Note. You must set the type for Expression 1 using the Expr1Type property before you can use this method.

Parameters

None.

Returns

A reference to a blank QueryExpression object.

See Also

Chapter 24, Query Classes , Expr1Expression, page 1383, Chapter 24, Query Classes , AddExpr1Field, page 1380, Chapter 24, Query Classes , Expr1Type, page 1383, Chapter 24, Query Classes , QueryExpression Class, page 1395.

AddExpr1Field

Syntax

AddExpr1Field(*QueryRecordAlias, FieldName*)

Description

The **AddExpr1Field** method returns a reference to a new a QueryField object to be used as a field for Expression 1. You can then use this object to specify details about the expression using the methods and properties of the QueryField Class.

Note. You must set the type for Expression 1 using the Expr1Type property before you can use this method.

Parameters

Parameter	Description
QueryRecordAlias	Specify the alias of the QueryRecord that contains the QueryField that you want to use.
FieldName	Specify the name of the QueryField in the QueryRecord that you want to use.

Returns

A reference to a blank QueryField object.

See Also

Chapter 24, Query Classes , Expr1Field, page 1383, Chapter 24, Query Classes , AddExpr1Expression, page 1379, Chapter 24, Query Classes , Expr1Type, page 1383, Chapter 24, Query Classes , QueryField Class, page 1368.

AddExpr2Expression

Syntax

AddExpr2Expression()

Description

The **AddExpr2Expression** method returns a reference to a new a QueryExpression object to be used as an expression for Expression 2. You can then use this object to specify details about the expression using the methods and properties of the QueryExpression Class.

Parameters

None.

Returns

A reference to a blank QueryExpression object.

See Also

Chapter 24, Query Classes , QueryExpression Class, page 1395.

AddExpr2Field1

Syntax

AddExpr2Field1(*QueryRecordAlias, FieldName*)

Description

Expression 2 has two field expressions because when the Between relational operator is used in a criteria, there can be two expression fields. The **AddExpr2Field1** method returns a reference to a new QueryField object to be used as a *field 1* for Expression 2. You can then use this object to specify details about the expression using the methods and properties of the QueryField Class.

Parameters

Parameter	Description
QueryRecordAlias	Specify the alias of the QueryRecord that contains the QueryField that you want to use.
FieldName	Specify the name of the QueryField in the QueryRecord that you want to use.

Returns

A reference to a blank QueryField object.

See Also

Chapter 24, Query Classes , QueryRecord Class, page 1363, Chapter 24, Query Classes , QueryField Class, page 1368.

AddExpr2Field2

Syntax

AddExpr2Field2(*QueryRecordAlias, FieldName*)

Description

Expression 2 has two field expressions because when the Between relational operator is used in a criteria, there can be two expression fields. The **AddExpr2Field2** method returns a reference to a new a QueryField object to be used as a *field 2* for Expression 2. You can then use this object to specify details about the expression using the methods and properties of the QueryField Class.

Parameters

Parameter	Description
QueryRecordAlias	Specify the alias of the QueryRecord that contains the QueryField that you want to use.
FieldName	Specify the name of the QueryField in the QueryRecord that you want to use.

Returns

A reference to a blank QueryField object.

See Also

Chapter 24, Query Classes , QueryRecord Class, page 1363, Chapter 24, Query Classes , QueryField Class, page 1368.

AddExpr2List

Syntax

```
AddExpr2List()
```

Description

Expression 2 can have a list of values when the Operator is of type In List (or Not In List). The **AddExpr2List** method returns a reference to a new QueryList, which can be used to add a list of values to the criteria.

Parameters

None.

Returns

A reference to a blank QueryList object.

See Also

Chapter 24, Query Classes , QueryList Class, page 1399.

AddExpr2Subquery

Syntax

```
AddExpr2Subquery()
```

Description

The **AddExpr2Subquery** method is used to create a subquery for Expression2. This method returns a new QuerySelect object you can use to specify details about the new subquery.

Warning! The new subquery created with this method replaces any existing subquery (for this criteria), destroying any existing properties or values.

Parameters

None.

Returns

A reference to a new QuerySelect object.

See Also

Chapter 24, Query Classes , QuerySelect Class, page 1346.

QueryCriteria Class Properties

In this section, we discuss the QueryCriteria class properties. The properties are discussed in alphabetical order.

Expr1Expression

Description

This property returns a reference to the QueryExpression object that's used as Expression 1.

This property is valid only when Expression 1 exists as an expression. If you want to add an expression for Expression 1, use the AddExpr1Expression method.

This property is read-only.

See Also

Chapter 24, Query Classes , AddExpr1Expression, page 1379.

Expr1Field

Description

This property returns a reference to the QueryField object that's used as Expression 1.

This property is valid only when Expression 1 exists as a field. You can then use the QueryField Class methods and property to manipulate this object.

If you want to add a field for Expression 1, use the AddExpr1Field method.

This property is read-only.

See Also

Chapter 24, Query Classes , QueryField Class, page 1368, Chapter 24, Query Classes , AddExpr1Field, page 1380.

Expr1Type

Description

This property returns or sets the type for Expression 1.

Note. You *must* set the type of expression for every new criteria.

The values are:

Numeric Value	Constant Value	Description
2	%Query_ExprField	Field
3	%Query_Expression	Expression

This property is read-write.

Example

The following is used to test the expression to determine the property to use to retrieve it:

```
&MyExpr1 = &MyQueryColl.Next();
If &MyExpr1.Expr1Type = %Query_ExprField Then /* Expression is a Field */

   &OldValue = &MyExpr1.Expr1Field;
   /* do processing */

Else  /* Expression 1 is an expression */

   &OldValue = &MyExpr1.Expr1Expression;
   /* Do processing */

End-if;
```

The following is an example showing how to add a field for Expression 1.

```
/* add a new criteria */
&MyCriteria = &MyQuery.AddCriteria();

/* set the type of the first expression to be a field */
&MyCriteria.Expr1Type = %Query_ExprField;

/* add the field EMPLID from the ABSENCE_HIST record */
&MyField = &MyCriteria.AddExpr1Field("A", "EMPLID");
```

Expr2Constant1

Description

If the Between relational operator is used in the criteria, there can be two constants for Expression 2. This property returns or sets the constant value for the first constant for Expression 2. This property takes a string value.

This property is valid only when Expression 2 is defined as a constant.

This property is read-write.

Expr2Constant2

Description

If the Between relational operator is used in the criteria, there can be two constants for Expression 2. This property returns or sets the constant value for the second constant for Expression 2. This property takes a string value.

This property is valid only when Expression 2 is defined as a constant.

This property is read-write.

Expr2Expression1

Description

If the Between relational operator is used in the criteria, there can be two expressions for Expression 2. This property returns a reference to the first QueryExpression object that's used as Expression 2.

This property is valid only when Expression 2 exists as an expression. To add an expression for Expression 2, use the AddExpr2Expression method.

This property is read-only.

See Also

Chapter 24, Query Classes , AddExpr2Expression, page 1380.

Expr2Expression2

Description

If the Between relational operator is used in the criteria, there can be two expressions for Expression 2. This property returns a reference to the second QueryExpression object that's used as Expression 2.

This property is valid only when Expression 2 exists as an expression. To add an expression for Expression 2, use the AddExpr2Expression method.

This property is read-only.

See Also

Chapter 24, Query Classes , AddExpr2Expression, page 1380.

Expr2Field1

Description

If the Between relational operator is used in the criteria, there can be two fields for Expression 2. This property returns a reference to the first QueryField object that's used as Expression 2.

This property is only valid when Expression 2 exists as a field. You can then use the QueryField Class methods and property to manipulate this object.

To add a field for Expression 2, use the AddExpr2Field1 method.

This property is read-only.

See Also
Chapter 24, Query Classes , AddExpr2Field1, page 1381, Chapter 24, Query Classes , QueryField Class, page 1368.

Expr2Field2

Description

If the Between relational operator is used in the criteria, there can be two fields for Expression 2. This property returns a reference to the second QueryField object that's used as Expression 2.

This property is valid only when Expression 2 exists as a field. You can then use the QueryField Class methods and property to manipulate this object.

To add a field for Expression 2, use the AddExpr2Field2 method.

This property is read-only.

See Also
Chapter 24, Query Classes , AddExpr2Field2, page 1381, Chapter 24, Query Classes , QueryField Class, page 1368.

Expr2List

Description

This property returns a reference to the QueryList object of Expression 2 that's used when the Operator is of type In List (or Not In List).

This property is read-only.

Expr2Subquery

Description

This property returns a reference to the QuerySelect object that's used as a subquery for Expression 2.

This property is valid only when Expression 2 exists as a subquery. To add a subquery for Expression 2, use the AddExpr2Subquery method.

This property is read-only.

See Also
Chapter 24, Query Classes , AddExpr2Subquery, page 1382.

Expr2Type

Description

This property returns or sets the type for Expression 2. The following table lists all of possible values for this property. However, the values for this property are dependent upon the Operator property.

This property is read-write.

See Chapter 24, Query Classes , Operator, page 1390 and Chapter 24, Query Classes , Working With Criteria and Expressions , page 1309.

You can use either a constant or numeric value for this property. The values are:

Numeric Value	Constant Value	Description
1	%Query_ExprConstant	Constant
2	%Query_ExprField	Field
3	%Query_Exprression	Expression
4	%Query_ExprSubQuery	Subquery
5	%Query_ExprList	List
6	%Query_ExprCurDt	Current date
7	%Query_ExprTree	Tree
8	%Query_ExprBind	Bind
9	%Query_ExprBothConst	The criterion's operator is Between and both values on the right-hand side are constants. (Const-Const)
10	%Query_ExprConstFld	The criterion's operator is Between, the first value on right-hand side is a constant and the second value is a field. (Const-Field)
11	%Query_ExprConstExpr	The criterion's operator is Between, the first value on right-hand side is a constant and the second value is an expression. (Const-Expr)
12	%Query_ExprFieldConst	The criterion's operator is Between, the first value on right-hand side is a field and the second value is a constant. (Field-Const)
13	%Query_ExprBothFld	The criterion's operator is Between and both values on the right-hand side are constants. (Field-Field)
14	%Query_ExprFldExpr	The criterion's operator is Between, the first value on right-hand side is a field and the second value is an expression. (Field-Expr)
15	%Query_ExprExprConst	The criterion's operator is Between, the first value on right-hand side is an expression and the second value is a constant. (Expr-Const)
16	%Query_ExprExprFld	The criterion's operator is Between, the first value on right-hand side is an expression and the second value is a field. (Expr-Field)
17	%Query_ExprBothExpr	The criterion's operator is Between and both values on the right-hand side are expressions. (Expr-Expr)

The following table describes how to access or change Expression 2 depending on the Expression2 Type.

Expression2 Type	Method or Property for Changing the Expression	Method or Property for Accessing the Expression
Constant	Expr2Constant	Expr2Constant
Field	AddExpr2Field()	Expr2Field
Expression	AddExpr2Expression()	Expr2Expression
In List	AddExpr2List	Expr2List
In Tree	AddExpr2Expression	Expr2Expression
Subquery	AddExpr2Subquery()	Expr2Subquery
Const-Const	Expr2Constant, Expr2Constant	Expr2Constant, Expr2Constant
Const-Field	Expr2Constant, AddExpr2Field()	Expr2Constant, Expr2Field
Const-Expr	Expr2Constant, AddExpr2Expression()	Expr2Constant, Expr2Expression
Field-Const	AddExpr2Field(), Expr2Constant	Expr2Field, Expr2Constant
Field-Field	AddExpr2Field(), AddExpr2Field()	Expr2Field, Expr2Field
Field-Expr	AddExpr2Field(), AddExpr2Expression()	Expr2Field, Expr2Expression
Expr-Const	AddExpr2Expression(), Expr2Constant	Expr2Expression, Expr2Constant

Expression2 Type	Method or Property for Changing the Expression	Method or Property for Accessing the Expression
Expr-Field	AddExpr2Expression(), AddExpr2Field()	Expr2Expression, Expr2Field
Expr-Expr	AddExpr2Expression(), AddExpr2Expression()	Expr2Expression, Expr2Expression

Example

The following is used to test the expression to determine the property to use to retrieve it:

```
&MyExpr2 = &MyQueryColl.Next();

If &MyExpr2.Expr2Type = %Query_ExprConstant Then /* Expression is a constant */

   &OldValue = &MyExpr2.Expr2Constant;
   /* do processing */

End-if;
```

The following is an example showing how to add a field for Expression 1.

```
/* add a new criteria */
&MyCriteria = &MyQuery.AddCriteria();

/* set the type of the first expression to be a field */
&MyCriteria.Expr1Type = %Query_ExprField;

/* add the EMPLID field from the ABSENCE_HIST record whose record alias is A */
&MyField = &MyCriteria.AddExpr1Field("A", "EMPLID" );
```

Logical

Description

This property returns or sets the logical portion of a criteria.

Note. This property is valid only when there are more than one criteria for a query. Also, this property is required when there is more than one criteria for a query.

The values are:

Numeric Value	Constant Value	Description
1	%Query_CombAnd	Logical And for non-having criteria
2	%Query_CombOr	Or for non-having criteria

Numeric Value	Constant Value	Description
3	%Query_CombNotUsed	No logical operator. Used for the first non-having criteria.
4	%Query_CombHaveAnd	Logical And for having criteria
5	%Query_CombHaveOr	Logical Or for having Criteria
6	%Query_CombHaveNotUsed	No logical operator. Used for the first having criteria.

This property is read-write.

LParenLvl

Description

This property returns or sets the left parenthesis level used for grouping criteria. This property takes a numeric value.

This property is read-write.

Name

Description

This property returns the name of the Query Criteria, as a string.

This property is read-only.

Negation

Description

This property returns a Boolean value, indicating whether a criterion is negated: True if the criterion is negated, False if it isn't.

This property is read-only.

Operator

Description

This property returns or sets the operator for the criteria.

The value of this property determines the valid types of the Expression 2, set with the Expr2Type property.

This property is read-write.

See Chapter 24, Query Classes , Expr2Type, page 1386 and Chapter 24, Query Classes , Working With Criteria and Expressions , page 1309.

You can use either a constant or numeric value for this property. The values are:

Numeric Value	Constant Value	Description
1	%Query_CondNone	None (used for initializing a new criteria.)
2	%Query_CondEqual	Criteria's left-hand side is equal to right-hand side (= operator)
3	%Query_CondNotEqual	Criteria's left-hand side is not equal to right-hand side (<> operator)
4	%Query_CondGreaterThan	Criteria's left-hand side is greater than right-hand side (> operator)
5	%Query_CondNotGreaterThan	Criteria's left-hand side is not greater than right-hand side (<= operator)
6	%Query_CondLessThan	Criteria's left-hand side is less than right-hand side (< operator)
7	%Query_CondNotLessThan	Criteria's left-hand side is not less than right-hand side (>= operator)
8	%Query_CondInList	Criteria's left-hand side is in the given list (IN operator)
9	%Query_CondNotInList	Criteria's left-hand side is not in the given list (Not IN operator)
10	%Query_CondBetween	Criteria's left-hand side is between the two values of right-hand side (BETWEEN operator)
11	%Query_CondNotBetween	Criteria's left-hand side is not between the two values of right-hand side (BETWEEN operator)
12	%Query_CondExists	Criteria's left-hand side is the output of the subquery of right-hand side (EXISTS operator)
13	%Query_CondNotExists	Criteria's left-hand side doesn't exist in the output of the subquery of right-hand side (NOT EXISTS operator)
14	%Query_CondLike	Criteria's left-hand side is like (wildcard search) the right-hand side (LIKE operation)
15	%Query_CondNotLike	Criteria's left-hand side is not like (wildcard search) the right-hand side (NOT LIKE operation)
16	%Query_CondNull	Criteria's left-hand side is NULL (NULL operation)
17	%Query_CondNotNull	Criteria's left-hand side is not NULL (IS NOT NULL operation)
18	%Query_CondInTree	Criteria's left-hand side is from a list of nodes in Tree (IN operation)
19	%Query_CondNotInTree	Criteria's left-hand side is not from a list of nodes in Tree (NOT IN operation)
20	%Query_CondEffDtLessEqual	Criteria's left-hand side is an Effective Date and is less than or equal to the date on the right-hand side (<= operation)

Numeric Value	Constant Value	Description
21	%Query_CondEffDtGreaterEqual	Criteria's left-hand side is an Effective Date and is greater than or equal to the date on the right-hand side (>= operation)
22	%Query_CondEffDtLess	Criteria's left-hand side is an Effective Date and is less than the date on the right-hand side (< operation)
23	%Query_CondEffDtGreater	Criteria's left-hand side is an Effective Date and is greater than the date on the right-hand side (> operation)
24	%Query_CondFirstEffDt	Criteria's left-hand side is the first effective date (Function MIN())
25	%Query_CondLastEffDt	Criteria's left-hand side is the last effective date (Function MAX())
26	%Query_CondInTreeJoin	Criteria's left-hand side is an In Tree Join.

R1ExprNum

Description

This property returns or sets the expression number for the first expression of Expression 2. This property takes a numeric value.

This property is read-write.

R2ExprNum

Description

This property returns or sets the expression number for the second expression of Expression 2. This property takes a numeric value.

This property is read-write.

R1ExprType

Description

This property returns the expression type for the first expression of Expression 2. This property takes a numeric value and is the same range of values as the Expr2Type. It helps distinguish the type of an expression based on the value of Expr2Type. For instance, if the Expr2Type is Field-Expr, R1Expr2Type is of type Field and R2Expr2Type is of type Expression.

This property is read-only.

R2ExprType

Description

This property returns the expression type for the second expression of Expression 2. This property takes a numeric value and is the same range of values as the Expr2Type. It helps distinguish the type of an expression based on the value of Expr2Type. For instance, if the Expr2Type is Field-Expr, R1Expr2Type is of type Field and R2Expr2Type is of type Expression.

This property is read-only.

RParenLvl

Description

This property returns or sets the right parenthesis level used for grouping criteria. This property takes a numeric value.

This property is read-write.

QueryExpression Collection

A QueryExpression Collection is returned from the Expressions Query class property.

See Chapter 24, Query Classes , Expressions, page 1354.

QueryExpression Collection Methods

In this section, we discuss the QueryExpression collection methods. The methods are described in alphabetical order.

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryExpression object in the QueryExpression collection.

Parameters

None.

Returns

A reference to a QueryExpression object if successful, NULL otherwise.

Example

```
&MyQueryExpression = &MyCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the QueryExpression object that exists at the *number* position in the QueryExpression collection.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the QueryExpression object that you want returned.

Returns

A reference to a QueryExpression object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryExpressionColl.Count;
   &MyQueryExpression = &QueryExpressionColl.Item(&I);
   /* do processing */
End-For;
```

ItemByName

Syntax

ItemByName(*ExpressionName*)

Description

The **ItemByName** method returns the QueryExpression object in the QueryExpressionCollection with the given expression name.

Parameters

Parameter	Description
ExpressionName	The name of the required expression.

Returns

A reference to a QueryExpression object if successful, NULL otherwise.

Example

```
&QryExpr = &QryExprCol.Item("Exp-6");
```

Next

Syntax

Next()

Description

The **Next** method returns the next QueryExpression object in the QueryExpression collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryExpression object if successful, NULL otherwise.

Example

```
&MyQueryExpression = &MyCollection.Next();
```

QueryExpression Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of QueryExpression objects in the QueryExpression Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryExpression Class

A QueryExpression object is return by the following:

- The AddExpression Query class method.
- The First, Item and Next methods of the QueryExpression Collection.

- All of the QueryCriteria Class Methods.
- The Expr1Expression and Expr2Expression1 QueryCriteria properties.

See Chapter 24, Query Classes , AddExpression, page 1348; Chapter 24, Query Classes , QueryExpression Collection, page 1393; Chapter 24, Query Classes , QueryCriteria Class Methods , page 1379; Chapter 24, Query Classes , Expr1Expression, page 1383 and Chapter 24, Query Classes , Expr2Expression1, page 1385.

See Chapter 24, Query Classes , Working With Criteria and Expressions , page 1309.

QueryExpression Class Properties

In this section, we discuss the QueryExpression class properties. The properties are discussed in alphabetical order.

Aggregate

Description

This property specifies whether the expression is an aggregate function.

This property takes a Boolean value: True if the expression is an aggregate function, False, otherwise.

This property is read-write.

BindFlag

Description

This property specifies whether the expression contains a bind value, as Boolean value.

This property is mainly used while reading a query, to determine whether a Query Expression contains a bind value. It isn't necessary to set this value while saving a query to the database.

Values are:

Value	Description
False	Expression doesn't have a bind value
True	Expression has a bind value

This property is read-write.

Decimal

Description

This property returns or sets the decimal value of the expression.

This property is only valid with numeric fields.

This property is read-write.

ExpNum

Description

This property returns or sets the unique expression number, as a numeric value.

This property is read-write.

IsXlatExpression

Description

This property indicates whether the expression is for a translate field. This property takes a Boolean value: True, the expression is based on a translated field.

This property is read-only.

Length

Description

This property returns or sets the length of the expression, as a number.

This property is read-write.

Name

Description

This property returns the name of the expression, as a string.

This property is read-only.

OutputField

Description

This property returns the instance of the displayed expression field, as a QueryField. This property returns a NULL when the expression isn't used in a displayed (output) field.

This property is read-write.

RightExprFlag

Description

This property specifies whether an expression is used in the right-hand side of a criteria (that is, is it an Expr2 expression), as a Boolean value.

This property is typically used while reading a query to determine whether a Query Expression is used in the right-hand side of criteria. It's not necessary to set this value while saving a query to the database.

Values are:

Value	Description
False	Expression is not used in the right-hand side of the criteria
True	Expression is used in the right-hand side of the criteria

This property is read-write.

SelectedField

Description

This property returns the instance of the expression field, as a QueryField. This property returns a NULL when the expression is used only in the right-hand side of a criteria.

This property is read-write.

Text

Description

This property returns or sets the text of the expression, as a character string.

This property is read-write.

Using ORACLE Hints in Expressions

Oracle hints can be included in expressions using the following considerations:

- Expression containing a hint must begin with **/***+.
- Expression can only contain one hint. For example, only one set of **/* */** is allowed in each expression.
- Each **/*** must precede an ***/.**
- Each expression must contain a complete hint. For example, an expression can't have only **/*** or ***/**. Both must be in same expression.

Type

Description

This property returns or sets the field type of the expression.

You can specify either a constant or a numeric value. The values are:

Numeric Value	Constant Value	Description
0	%FieldType_Char	Character
1	%FieldType_LongChar	Long Character
2	%FieldType_Number	Number
3	%FieldType_SignedNumber	Signed number
4	%FieldType_Date	Date

Numeric Value	Constant Value	Description
5	%FieldType_Time	Time
6	%FieldType_DateTime	Datetime
7	%FieldType_Image	Image

This property is read-write.

QueryList Class

A QueryList Class is returned from the Expr2List property and AddExpr2List method of the QueryCriteria Class.

See Chapter 24, Query Classes , Expr2List, page 1386 and Chapter 24, Query Classes , AddExpr2List, page 1382.

QueryList Class Methods

In this section, we discuss the QueryList class methods. The methods are discussed in alphabetical order.

AddListValue

Syntax

AddListValue(*Value, IsPrompt*)

Description

The **AddListValue** method adds a new List Value into the QueryList instance.

Parameters

Parameter	Description
Value	Specify the string value to be added to the list
IsPrompt	Specify whether the string specified by *Value* is a bind variable. This parameter takes a Boolean value: True, *Value* is a bind variable.

Returns

A reference to a QueryListValue object if successful, NULL otherwise.

Example

```
&MyListValue = &MyList.AddListValue("1", False);
```

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryListValue object in the QueryList instance.

Parameters

None.

Returns

A reference to a QueryListValue object if successful, NULL otherwise.

Example

```
&MyListValue = &MyList.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns the QueryListValue object that exists at the *number* position in the QueryList instance.

Parameters

Parameter	Description
Number	Specify the position number in the QueryList object that you want returned.

Returns

A reference to a QueryListValue object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryList.Count;
   &QryListVal = & QueryList.Item(&I);
   /* do processing */
End-For;
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next QueryListValue object in the QueryList instance. This method should be called after calling First(), else it returns NULL.

Parameters

None.

Returns

A reference to a QueryListValue object if successful, NULL otherwise.

Example

```
&MyNextListValue = &MyList.Next();
```

QueryList Class Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of QueryListValue objects in the QueryList Collection, as a number.

This property is read-only.

Example

```
&ListValueCount = &MYList.Count;
```

QueryListValue Class

A QueryListValue instance is returned from the AddListValue, First, Item, or Next QueryList Class methods.

See Chapter 24, Query Classes , QueryList Class Methods , page 1399.

QueryListValue Class Properties

In this section, we discuss the QueryListValue class properties. These properties are discussed in alphabetical order.

IsPrompt

Description

This property indicates whether the value is a bind variable (such as :1). This property takes a Boolean value: True, this property is a bind variable.

This property is read-only.

Value

Description

This property returns a string for the value stored in the list.

This property is read-only.

QueryRecordHierarchy Collection

A QueryRecordHierarchy collection is returned from the RecordHierarchy QueryDBRecord property.

The order of each QueryRecordHierarchy object in the collection maps to the order of each tree node as it appears in the tree hierarchy from top to bottom.

See Chapter 24, Query Classes , RecordHierachy, page 1419.

QueryRecordHierarchy Collection Methods

In this section, we discuss the QueryRecordHierarchy collection methods. The methods are discussed in alphabetical order.

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryRecordHierarchy object in the QueryRecordHierarchy collection.

Parameters

None.

Returns

A reference to a QueryRecordHierarchy object if successful, NULL otherwise.

Example

```
&MyQueryRecordHierarchy = &MyCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the QueryRecordHierarchy object that exists at the *number* position in the QueryRecordHierarchy collection.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the QueryRecordHierarchy object that you want returned.

Returns

A reference to a QueryRecordHierarchy object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryRecordHierarchyColl.Count;
   &MyQueryRecordHierarchy = &QueryRecordHierarchyColl.Item(&I);
   /* do processing */
End-For;
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the QueryRecordHierarchy object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing QueryRecordHierarchy within the QueryRecordHierarchy collection. If you specify an invalid name, the object is NULL.

Returns

A reference to a QueryRecordHierarchy object if successful, NULL otherwise.

Example

```
&MyQueryRecordHierarchy = &MyCollection.ItemByName("PHONELIST");
```

Next

Syntax

Next()

Description

The **Next** method returns the next QueryRecordHierarchy object in the QueryRecordHierarchy collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryRecordHierarchy object if successful, NULL otherwise.

Example

```
&MyQueryRecordHierarchy = &MyCollection.Next();
```

QueryRecordHierarchy Collection Property

In this section, we discuss the QueryRecordHierarchy properties. These properties are discussed in alphabetical order.

Count

Description

This property returns the number of QueryRecordHierarchy objects in the QueryRecordHierarchy Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryRecordHierarchy Class

A reference to a QueryRecordHierarchy object is returned by the the First, Item, ItemByName, and Next QueryRecordHierarchy Collection methods.

Every QueryRecordHierarchy object returned from a collection represents a record node in the Record Hierarchy tree in Query tab of PeopleSoft Query.

The QueryRecordHierarchy object returned from the QueryField represents the prompt table for the record field.

See Chapter 24, Query Classes , QueryRecordHierarchy Collection, page 1402.

QueryRecordHierarchy Class Properties

In this section, we discuss the QueryRecordHierarchy class properties. The properties are discussed in alphabetical order.

Description

Description

This property returns a description of the record node as a string.

This property is read-only.

Level

Description

This property returns the level of the record node in the record hierarchy. 1 is the root node, 2 is a node beneath the root node, 3 is a child of that, and so on.

This property is read-only.

Name

Description

This property returns the name of the record node as a string.

This property is read-only.

ParentFlag

Description

This property returns the parent flag. The values are:

Value	Description
0	Record node contains no children nodes.
1	Record node contains children nodes

This property is read-only.

Query Metadata Collection

A Query Metadata collection is returned by the Metadata Query class property.

See Chapter 24, Query Classes , Metadata, page 1342.

See Chapter 24, Query Classes , Using Query Metadata , page 1311.

Query Metadata Collection Methods

In this section, we discuss the Query Metadata collection methods. These methods are discussed in alphabetical order.

First

Syntax

```
First()
```

Description

The **First** method returns the first Query Metadata object in the Query Metadata collection.

Parameters

None.

Returns

A reference to a Query Metadata object if successful, NULL otherwise.

Example

```
&MyMetadata = &MyCollection.First();
```

Item

Syntax

```
Item(number)
```

Description

The **Item** method returns the Query Metadata object that exists at the *number* position in the Query Metadata collection.

Parameters

Parameter	Description
number	Specify the position number in the collection of the Query Metadata object that you want returned.

Returns

A reference to a Query Metadata object if successful, NULL otherwise.

Example

```
For &I = 1 to &MetadataColl.Count;
   &MyMetadata = &MetadataColl.Item(&I);
   /* do processing */
End-For;
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the Query Metadata object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing Query Metadata within the Query Metadata collection. If you specify an invalid name, the object is NULL.

Returns

A reference to a Query Metadata object if successful, NULL otherwise.

Example

```
&MyMetadata = &MyQuery.Metadata.ItemByName("Descr");
```

Next

Syntax

Next()

Description

The **Next** method returns the next Query Metadata object in the Query Metadata collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a Query Metadata object if successful, NULL otherwise.

Example

```
&MyMetadata = &MyCollection.Next();
```

Query Metadata Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of Query Metadata objects in the Query Metadata Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Query Metadata Class

A Query Metadata object is returned from the Query Metadata Collection methods First, Item, ItemByName, or Next.

See Chapter 24, Query Classes , Query Metadata Collection, page 1406.

See Chapter 24, Query Classes , Using Query Metadata , page 1311.

Query Metadata Class Properties

In this section, we discuss the Query Metadata class properties. The properties are discussed in alphabetical order.

Name

Description

This property returns the name of each Query Metadata property as a string.

Values for this property are:

Value	Description
Descr	Description
LongDescr	Long description
Public/Private	Specifies if the query is public or private. If the query is private, the name of the owner is listed in Value.
LastUpdDttm	Last updated date and time
LastUpdOprId	The UserId of the user who updated the value last
Record	Record name. May be more than one.
Input Param	Input parameter. May be more than one.
Expression	Expression. May be more than one.
Field	Record field name for the output column. May be more than one.
Heading	Heading name for the output column. May be more than one.

This property is read-only.

Value

Description

This property returns the value for the Query Metadata property as a string.

This property is read-only.

QueryStatistics Class

The QueryStatistics class is used to view statistical information about a query's execution. It can be useful for query administration. You can view query statistics for a query before you save it to the database.

A QueryStatistics Class is returned from the QueryStatistics Query property.

See Chapter 24, Query Classes , QueryStatistics, page 1345.

QueryStatistics Class Properties

In this section, we discuss the QueryStatistics properties. The properties are discussed in alphabetical order.

AvgExecTime

Description

This property returns the average execution time for the query as a string.

This property is read-only.

AvgFetchTime

Description

This property returns the average fetch time for the query as a string.

This property is read-only.

AvgNumRows

Description

This property returns the average number of rows fetched for the query.

This property is read-only.

ExecCount

Description

This property returns the total number of times the query has been executed.

This property is read-only.

LastExecDtTm

Description

This property returns the last date and time the query was executed, as a string.

This property is read-only.

QuerySecurityProfile Class

The QuerySecurityProfile class is used to view the current user's security profile for PeopleSoft Query. This class doesn't contain any methods, and all the properties are read-only. An instance of this class is returned by the GetQuerySecurityProfile Session method.

See Chapter 24, Query Classes , GetQuerySecurityProfile, page 1320.

QuerySecurityProfile Class Properties

In this section, we discuss the QuerySecurityProfile properties. The properties are discussed in alphabetical order.

AllowAnyJoin

Description

This property indicates whether the user is allowed to define queries with any join. This property takes a Boolean value: True, the user can define such queries.

This property is read-only.

AllowDistinct

Description

This property indicates whether the user can define queries with a Distinct clause in a SELECT statement. This property takes a Boolean value: True, the user can define such queries.

This property is read-only.

AllowExpressions

Description

This property indicates whether the user is allowed to define expressions in queries. This property takes a Boolean value: True, the user can define such queries.

This property is read-only.

AllowSubqueries

Description

This property indicates whether the user is allowed to define criteria containing subqueries. This property takes a Boolean value: True, the user can define such queries.

This property is read-only.

AllowUnions

Description

This property indicates whether the user is allowed to define queries containing unions. This property takes a Boolean value: True, the user can define such queries.

This property is read-only.

ApprovePrivateQuery

Description

This property indicates whether the user is allowed to approve private queries. This property takes a Boolean value: True, the user can approve such queries. This property is meant for query administration.

This property is read-only.

ApprovePublicQuery

Description

This property indicates whether the user is allowed to approve public queries. This property takes a Boolean value: True, the user can approve such queries. This property is meant for query administration.

This property is read-only.

CanCreatePublic

Description

This property indicates whether the user can create public queries. This property takes a Boolean value: True, the user can create such queries.

This property is read-only.

CanCreateWorkFlow

Description

This property indicates whether the user can create public workflow queries. This property takes a Boolean value: True, the user can create such queries.

This property is read-only.

CanModifyQuery

Description

This property indicates whether the user can modify queries. This property takes a Boolean value: True, the user can modify queries.

This property is read-only.

CanRunQuery

Description

This property indicates whether the user can run queries. This property takes a Boolean value: True, the user can run queries.

This property is read-only.

CanRunToCrystal

Description

This property indicates whether the user can run queries to a Crystal report. This property takes a Boolean value: True, the user can run queries.

This property is read-only.

CanRunToExcel

Description

This property indicates whether the user can run queries to an Excel spreadsheet. This property takes a Boolean value: True, the user can run queries.

This property is read-only.

LimitUnapproved

Description

This property indicates whether there is a limit on the number of rows returned for unapproved queries. This property is meant for query administration. This property takes a Boolean value: True, limit the number of rows. The MaxUnapprovedRows property is active only if this property is specified as True.

This property is read-only.

See Also

Chapter 24, Query Classes , MaxUnapprovedRows, page 1414.

MaxInTreeCriteria

Description

This property indicates the maximum number of In Tree Criteria that can be used in the queries defined by the current user. This property takes a numeric value.

This property is read-only.

MaxJoins

Description

This property indicates the maximum number of joins allowed in the queries defined by the current user. This property takes a numeric value.

This property is read-only.

MaxRowsToFetch

Description

This property indicates the maximum number of rows to fetch for the current user when a query is executed. This property takes a numeric value.

This property is read-only.

MaxUnapprovedRows

Description

This property indicates the maximum number of rows that can be returned for unapproved queries. This property takes a numeric value. This property is meant for query administration. This property is active only if the property LimitUnapproved is True.

This property is read-only.

See Also

Chapter 24, Query Classes , LimitUnapproved, page 1413.

QueryDBRecord Collection

A QueryDBRecord collection is returned from the FindQueryDBRecords session method.

See Chapter 24, Query Classes , FindQueryDBRecords, page 1317.

QueryDBRecord Collection Methods

In this section, we discuss the QueryDBRecord collection methods. The methods are discussed in alphabetical order.

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryDBRecord object in the QueryDBRecord collection.

Parameters

None.

Returns

A reference to a QueryDBRecord object if successful, NULL otherwise.

Example

```
&MyQueryDBRecord = &MyCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the QueryDBRecord object that exists at the *number* position in the QueryDBRecord collection.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the QueryDBRecord object that you want returned.

Returns

A reference to a QueryDBRecord object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryDBRecordColl.Count;
   &MyQueryDBRecord = &QueryDBRecordColl.Item(&I);
   /* do processing */
End-For;
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the QueryDBRecord object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing QueryDBRecord within the QueryDBRecord collection. If you specify an invalid name, the object is NULL.

Returns

A reference to a QueryDBRecord object if successful, NULL otherwise.

Example

```
&MyQueryDBRecord = &MyCollection.ItemByName("PHONELIST");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next QueryDBRecord object in the QueryDBRecord collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryDBRecord object if successful, NULL otherwise.

Example

```
&MyQueryDBRecord = &MyCollection.Next();
```

QueryDBRecord Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of QueryDBRecord objects in the QueryDBRecord Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryDBRecord Class

A QueryDBRecord object is returned from the QueryDBRecord Collection methods First, Item, ItemByName, or Next.

See Chapter 24, Query Classes , QueryDBRecord Collection, page 1414.

QueryDBRecord Class Methods

In this section, we discuss the QueryDBRecord class methods. The methods are discussed in alphabetical order.

QueryDBRecordFieldByIndex

Syntax

QueryDBRecordFieldByIndex(*Index*)

Description

The **QueryDBRecordFieldByIndex** method returns the QueryDBRecordField object that exists at the *index* position in the QueryDBRecordField collection.

Parameters

Parameter	Description
Index	Specify the position number in the collection of the QueryDBRecordField object that you want returned.

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

See Also

Chapter 24, Query Classes , QueryDBRecordFieldByName, page 1417, Chapter 24, Query Classes , QueryDBRecordField Class, page 1422.

QueryDBRecordFieldByName

Syntax

QueryDBRecordFieldByName(*Name*)

Description

The **QueryDBRecordFieldByName** method returns the QueryDBRecordField object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of the QueryDBRecordField object that you want returned.

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

See Also

Chapter 24, Query Classes , QueryDBRecordFieldByIndex, page 1417, Chapter 24, Query Classes , QueryDBRecordField Class, page 1422.

QueryDBRecord Class Properties

In this section, we discuss the QueryDBRecord class properties. The properties are discussed in alphabetical order.

Description

Description

This property returns the description of the QueryDBRecord as a string.

This property is read-only.

Name

Description

This property returns the name of the QueryDBRecord as a string.

This property is read-only.

QueryDBRecordFields

Description

This property returns a reference to a QueryDBRecordField Collection.

This property is read-only.

See Also

Chapter 24, Query Classes , QueryDBRecordField Collection, page 1419.

RecordHierachy

Description

This property returns a reference to a QueryRecordHierarchy Collection.

The record hierarchy is not related to the query tree hierarchy shown when viewing access groups. Instead, it reflects an actual relationship between the record components, as defined in PeopleSoft Application Designer using the Parent Record Name feature.

This property is read-only.

See Also

Chapter 24, Query Classes , QueryRecordHierarchy Collection, page 1402, .

QueryDBRecordField Collection

A QueryDBRecordField collection is returned from the QueryDBRecordFields QueryDBRecord property.

See Chapter 24, Query Classes , QueryDBRecordFields, page 1418.

QueryDBRecordField Collection Methods

In this section, we discuss the QueryDBRecordField collection methods. The methods are discussed in alphabetical order.

First

Syntax

```
First()
```

Description

The **First** method returns the first QueryDBRecordField object in the QueryDBRecordField collection.

Parameters

None.

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

Example

```
&MyQueryDBRecordField = &MyCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the QueryDBRecordField object that exists at the *number* position in the QueryDBRecordField collection.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the QueryDBRecordField object that you want returned.

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

Example

```
For &I = 1 to &Coll.Count;
   &MyQueryDBRecordField = &Coll.Item(&I);
   /* do processing */
End-For;
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the QueryDBRecordField object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing QueryDBRecordField within the QueryDBRecordField collection. If you specify an invalid name, the object is NULL.

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

Example

```
&MyQueryDBRecordField = &MyCollection.ItemByName("PHONELIST");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next QueryDBRecordField object in the QueryDBRecordField collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryDBRecordField object if successful, NULL otherwise.

Example

```
&MyQueryDBRecordField = &MyCollection.Next();
```

Sort

Syntax

```
Sort(SortCriteria)
```

Description

Use the **Sort** method to sort the fields within the QueryDBRecordField collection, based on the sort criteria specified with the method.

Parameters

Parameter	Description
SortCriteria	Specify the sort order for the list. This parameter can take either a constant or numeric value. The values are:

Numeric Value	Constant Value	Description
1	%Query_SortNameAsc	Sort database fields in ascending order based on field name.
2	%Query_SortNameDesc	Sort database fields in descending order based on field name.
3	%Query_SoftFldNumAsc	Sort database fields in ascending order based on field number.
4	%Query_SortFldNumDesc	Sort database fields in descending order based on field number.

Returns

None.

QueryDBRecordField Collection Property

In this section, we discuss the Count property.

Count

Description

This property returns the number of QueryDBRecordField objects in the QueryDBRecordField Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryDBRecordField Class

A QueryDBRecordField object is from the QueryDBRecord Collection methods First, Item, ItemByName or Next.

See Chapter 24, Query Classes , QueryDBRecord Collection, page 1414.

QueryDBRecordField Class Properties

In this section, we discuss the QueryDBRecordField class properties. The properties are discussed in alphabetical order.

Decimal

Description

This property returns the decimal positions for a field. This indicates how many numbers are allowed on the right side of the decimal.

This property is read-only.

Example

```
&FldDecs = &QryDBRcdFlds.Decimal;
```

Description

Description

This property returns the long description of the field as a string.

This is property is read-only.

Flag

Description

This property returns the Use and Edit Flags of the field, which are set when the Field is defined in PeopleSoft Application Designer, as a numeric value.

Values for Use Flags are:

Value	Description
1	Key
2	Duplicate Key
4	System
8	Audit Field Add
16	Alternate Key
32	List Item
64	Descending Key
128	Audit Field Change
1024	Audit Field Delete
2048	Search Item
32768	Auto Update

Values for Edit Flags are:

Value	Description
256	Required
512	Edit Translate
4096	Date Range Edit
8192	Yes/No Table Edit
16384	Edit Table
262144	From Search Field
524288	Through Search Field

Value	Description
8388608	Use Default Label Flag
16777216	Default Search Field

This is a read-only property.

Format

Description

This property returns the field format for a field. Values are:

Value	Description
1	No format
2	Name
3	Phone Number North America
4	Zip/Postal Code North America
5	Social Security Number (SSN)
6	Uppercase
7	Mixed case
8	Raw binary
9	Numbers only
10	Canadian Social Insurance Number (SIN)
11	Phone Number International
12	Zip/Postal Code International
13	Seconds
14	Microseconds
15	Custom

This property is read-only.

Length

Description

This property returns the length of the field as a number.

This property is read-only.

LongName

Description

This property returns the long name of the field as a string.

This property is read-only.

LookupTableName

Description

If the field has a lookup table associated with it, this property returns the name of that Lookup Table, else it returns an empty string.

This is a read-only property.

LookupTableRecord

Description

If the field has a lookup table associated with it, this property returns the instance of the QueryDBRecord for that Lookup Table, else it returns NULL.

This is a read-only property.

Name

Description

This property returns the name of the field as a string.

This property is read-only.

Shortname

Description

This property returns the short name of the field as a string.

This property is read-only.

Type

Description

This property returns the type of the field. You can specify either a constant or a numeric value for this property. The values are:

Numeric Value	Constant Value	Description
0	%FieldType_Char	Character
1	%FieldType_LongChar	Long Character

Numeric Value	Constant Value	Description
2	%FieldType_Number	Number
3	%FieldType_SignedNumber	Signed number
4	%FieldType_Date	Date
5	%FieldType_Time	Time
6	%FieldType_DateTime	Datetime
7	%FieldType_Image	Image

This property is read-only.

QueryPrompt Collection

A QueryPrompt collection is returned from the Prompts and RuntimePrompts Query class properties.

See Chapter 24, Query Classes , Prompts, page 1343 and Chapter 24, Query Classes , RunTimePrompts, page 1345.

QueryPrompt Collection Methods

In this section, we discuss the QueryPrompt collection methods. The methods are discussed in alphabetical order.

First

Syntax

First()

Description

The **First** method returns the first QueryPrompt object in the QueryPrompt collection.

Parameters

None.

Returns

A reference to a QueryPrompt object if successful, NULL otherwise.

Example

```
&MyQueryPrompt = &MyCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the QueryPrompt object that exists at the *number* position in the QueryPrompt collection.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the QueryPrompt object that you want returned.

Returns

A reference to a QueryPrompt object if successful, NULL otherwise.

Example

```
For &I = 1 to &QueryPromptColl.Count;
   &MyQueryPrompt = &QueryPromptColl.Item(&I);
   /* do processing */
End-For;
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the QueryPrompt object with the name *Name*.

Parameters

Parameter	Description
Name	Specify the name of an existing QueryPrompt within the QueryPrompt collection. If you specify an invalid name, the object is NULL.

Returns

A reference to a QueryPrompt object if successful, NULL otherwise.

Example

```
&MyQueryPrompt = &MyCollection.ItemByName("PHONELIST");
```

Next

Syntax

Next()

Description

The **Next** method returns the next QueryPrompt object in the QueryPrompt collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Parameters

None.

Returns

A reference to a QueryPrompt object if successful, NULL otherwise.

Example

```
&MyQueryPrompt = &MyCollection.Next();
```

QueryPrompt Collection Property

In this section, we discuss the QueryPrompt collection properties. The properties are discussed in alphabetical order.

Count

Description

This property returns the number of QueryPrompt objects in the QueryPrompt Collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

QueryPrompt Class

A reference to a QueryPrompt is returned by the following:

- The First, Item, ItemByName, and Next methods of a QueryPrompt Collection.
- The AddPrompt Query class method.

See Chapter 24, Query Classes , QueryPrompt Collection, page 1426 and Chapter 24, Query Classes , AddPrompt, page 1327.

See .

QueryPrompt Properties

In this section, we discuss the QueryPrompt properties. The properties are discussed in alphabetical order.

EditType

Description

This property returns or sets the edit type for the field. This property takes a number value. The values are:

Value	Description
0	No table edit
16384	Prompt table
512	Translate table
8192	Yes/No

This property is read-write.

FieldDecimal

Description

This property returns or sets the decimal value for the field.

This property is only valid with number fields.

This property is read-write.

FieldFormat

Description

This property returns the field format for a field. This property takes a number value. Values are:

Value	Description
1	No format
2	Name
3	Phone Number North America
4	Zip/Postal Code North America
5	Social Security Number (SSN)
6	UpperCase
7	Mixed case
8	Raw binary

Value	Description
9	Numbers only
10	Canadian Social Insurance Number (SIN)
11	Phone Number International
12	Zip/Postal Code International
13	Seconds
14	Microseconds
15	Custom

This property is read-only.

FieldLength

Description

This property returns or sets the field length.

This property is read-write.

FieldName

Description

This property returns or sets the field name used with the prompt.

This property is read-write.

FieldType

Description

This property returns or sets the field type of the field used with the prompt.

This property returns the type of the field. You can specify either a constant or a numeric value. The values are:

Numeric Value	Constant Value	Description
0	%FieldType_Char	Character
1	%FieldType_LongChar	Long Character
2	%FieldType_Number	Number
3	%FieldType_SignedNumber	Signed number
4	%FieldType_Date	Date
5	%FieldType_Time	Time
6	%FieldType_DateTime	Datetime
7	%FieldType_Image	Image

This property is read-write.

HeadingText

Description

This property returns or sets the heading text for the prompt field.

This property is read-write.

HeadingType

Description

This property returns or sets the heading type for the query field. This property takes either a constant or numeric value. The values are:

Numeric Value	Constant Value	Description
1	%Query_HdgNone	Query field has no heading.
2	%Query_HdgText	Query field has a text heading.
3	%Query_HdgRftShort	Query field uses the short RFT heading.
4	%Query_HdgRftLong	Query fields uses the long RFT heading.

This property is read-write.

LangCount

Description

This property returns the total count of the language records for the current prompt.

This property is read-only.

Name

Description

This property returns a string containing the Prompt name. When an existing query is read, this name is the same as the Field Name. When a new prompt is added using AddPrompt, this is the *Name* parameter used with that method.

This property is read-only.

PromptRecordFieldName

Description

When you have more than one field with the same name in the prompt collection, the system generates a unique prompt field name for each repeated field. The generated names are of the form **Bind1**, **Bind2**, and so on. This property returns the unique name for the prompt record field.

This property is read-only.

PromptTable

Description

This property returns or sets the prompt table name for the prompt field.

This property is read-write.

UniquePromptName

Description

This property returns or sets the unique prompt name for the prompt field.

This property is read-write.

UseCount

Description

This property returns the total count of the usage of the current prompt.

This property is read-only.

The following are examples of the usual actions that you perform using the Query classes.

Query Classes Examples

The following are examples of the usual actions that you perform using the Query classes.

Creating a New Query

In this example, you are creating a new query, adding a record and two fields. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &aQuery, &aQrySelCol;
Local ApiObject &COLL, &ERROR;
Local String &TEXT;
Local Session &MySession;
Local Record &aQryRcd;
Local Field &aQryFld;

&MySession = %Session;

If &MySession <> Null Then

   &aQuery = &MySession.GetQuery();

   &aQuery.Create("TEST1", False, %Query_Query, "PIA Test 1", "Creating Test ⇒
Query 1 from PIA Page");
```

```
      &aQrySel = &aQuery.AddQuerySelect();

      &aQryRcd = &aQrySel.AddQueryRecord("ABSENCE_HIST");
      &aQryFld = &aQrySel.AddQuerySelectedField("ABSENCE_HIST", "A", "EMPLID", "ID");

      If &aQryFld <> Null Then
            &aQryFld.ColumnNumber = 1;
            &aQryFld.HeadingType = %Query_HdgRftShort;
         End-If;

      &Rslt = &aQuery.Save();

      If &Rslt <> 0 Then
         /* save didn't complete */
         &COLL = &MySession.PSMessages;

         For &I = 1 to &COLL.Count
            &ERROR = &COLL.Item(&I);
            &TEXT = &ERROR.Text;
            /* do error processing */
         End-For;

         &COLL.DeleteAll();
      End-if;
       /* error processing for not getting a session */
   End-if;
```

To create a new query:

1. Get a session object.

   Before you can create a query, you have to get a session object. The session controls access to the query, provides error tracing, enables you to set the runtime environment, and so on. Then this program checks to verify that the session object is valid.

   ```
   &MySession = %Session;

   &aQuery = &MySession.GetQuery();

   If &MySession <> Null Then
   ```

2. Create the query.

   Use the **Create** method to create the query. This query is a private query, of type query.

   ```
   &aQuery.Create("TEST1", False, %Query_Query, "Test 1", "Creating Test Query");
   ```

3. Add a QuerySelect.

The QuerySelect contains the main query statement for the query. There can be multiple QuerySelect objects for Queries that involve unions or subqueries. Each select (or union or subquery) consists of QueryRecords, QueryOutputFields, QuerySelectedFields, and QueryCriteria and is treated as a child of the MAIN select statement.

```
&aQrySel = &aQuery.AddQuerySelect();
```

4. Add a record and a field.

   The AddQueryRecord method adds a query record to the query. The AddQuerySelectedField adds a field, using the record alias "A". The ID is what gets displayed in the heading for the query.

```
&aQryRcd = &aQrySel.AddQueryRecord("ABSENCE_HIST");
&aQryFld = &aQrySel.AddQuerySelectedField("ABSENCE_HIST", "A", "EMPLID", "ID");
```

5. Make the field an output field.

   The field was added as a selected field. By setting the ColumnNumber to a number greater than one, the field is now an output field. The text that's displayed in the heading comes from the RFT short description of the field.

```
If &aQryFld <> Null Then
    &aQryFld.ColumnNumber = 1;
    &aQryFld.HeadingType = %Query_HdgRftShort;
 End-If;
```

6. Save the data.

   When you execute the Save method, the new query is saved to the database.

```
&Rslt = &aQuery.Save();

If &Rslt <> 0 Then
```

   The **Save** method returns a numeric value: 0 if successful. You can use this value to do error checking.

7. Check Errors.

   You can check if there were any errors using the PSMessages property on the session object.

```
   /* save didn't complete */
   &COLL = &MySession.PSMessages;
   For &I = 1 to &COLL.Count
      &ERROR = &COLL.Item(&I);
      &TEXT = &ERROR.Text;
      /* do error processing */
   End-For;
   &COLL.DeleteAll();
End-if;
```

   If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, you may want to delete it from the PSMessages collection.

Adding Criteria

In this example, you are accessing an existing query, then adding criteria both as part of the query as well as part of a subquery. The SQL statement created by this subquery is as follows:

```
SELECT RECNAME, RECDESCR, RELLANGRECNAME, PARENTRECNAME, DESCRLONG from PSRECDEFN ⇒
where RECNAME IN (select OBJECTVALUE1 from PSPROJECTITEM where PROJECTNAME =⇒
 'PPLTOOLS') AND RECTYPE = 0 order by RECNAME
```

The following is the complete code sample: the steps explain each line.

```
Local ApiObject &MyQuery, &MainQrySel, &Criteria1, &MyCrit2Expr2, &MyCriteria2;
Local ApiObject &SubQrySel, &SubQryCrit1, &SubQryExpr1, &MyCrit2Expr2;
Local Record &SubQryRec;
Local Session = &MySession;
Local ApiObject &COLL, &ERROR;
Local String &TEXT;

&MySession = %Session;

If &MySession <> Null Then

&MyQuery = &MySession.GetQuery();

&MyQuery.Open("Table", False, True);

&MainQrySel = &MyQuery.QuerySelect;

/* Adding query recory, adding fields, then making selected fields output fields *⇒
/

&MainRec = &MainQrySel.AddQueryRecord("PSRECDEFN");

&QryFld = &MainQrySel.AddQuerySelectedField(&MainRec.Name, &MainRec.RecordAlias,⇒
 "RECNAME", "Record Name");
&QryFld.ColumnNumber = 1;
&QryFld.OrderByNumber = 1;

&QryFld = &MainQrySel.AddQuerySelectedField(&MainRec.Name, &MainRec.RecordAlias,⇒
 "RECDESCR", "Record Descr");
&QryFld.ColumnNumber = 2;
&QryFld.OrderByNumber = 2;

&QryFld = &MainQrySel.AddQuerySelectedField(&MainRec.Name, &MainRec.RecordAlias,⇒
 "RELLANGRECNAME", "Record Lang Rec");
&QryFld.ColumnNumber = 3;
&QryFld.OrderByNumber = 3;

&QryFld = &MainQrySel.AddQuerySelectedField(&MainRec.Name, &MainRec.RecordAlias,⇒
 "PARENTRECNAME", "Parent Record Name");
&QryFld.ColumnNumber = 4;

&QryFld = &MainQrySel.AddQuerySelectedField(&MainRec.Name, &MainRec.RecordAlias,⇒
 "DESCRLONG", "Long Descr");
&QryFld.ColumnNumber = 5;
```

```
&QryFld = &MainQrySel.AddQuerySelectedField(&MainRec.Name, &MainRec.RecordAlias,⇒
 "EMPLID", "ID");
&QryFld.ColumnNumber = 6;

/* adding first criteria */
&Criteria1 = &MainQrySel.AddCriteria("FirstCriteria");

/* First criteria will not have any logical AND/OR */
&Criteria1.Logical = %Query_CombNotUsed;

&Criteria1.Expr1Type = %Query_ExprField;
&Criteria1.AddExpr1Field(&MainRec.RecordAlias, "RECNAME");

/* So that the criteria is constructed as - RECNAME IN (...)*/
&Criteria1.Operator = %Query_CondInList;
&Criteria1.Expr2Type = %Query_ExprSubQuery;

&SubQrySel = &Criteria1.AddExpr2SubQuery();
&SubQryRec = AddQueryRecord("PSPROJECTITEM");
&SubQryFld1 = &SubQrySel.AddQuerySelectedField(&SubQryRec.Name, &SubQryRec.Record⇒
Alias, "OBJECTVALUE1", "Join Object")
&SubQryFld1.ColumnNumber = 1;

/* Need criteria - PROJECTNAME = 'PPLTOOLS'  - in the subquery */
&SubQryCrit1 = &SubQrySel.AddCriteria("FirstSubCrit");

/* First criteria will not have any logical AND/OR */
&SubQryCrit1.Logical = %Query_CombNotUsed;

&SubQryCrit1.Expr1Type = %Query_ExprField;
&SubQryCrit1.AddExpr1Field(&SubQryRec.RecordAlias, "PROJECTNAME");
&SubQryCrit1.Operator = %Query_CondEqual;

/* So that the criteria is constructed as - PROJECTNAME = 'PPLTOOLS'*/
&SubQryCrit1.Expr2Type = %Query_ExprConstant;
&SubQryExpr1 = &SubQryCrit1.AddExpr2Expression();
&SubQryExpr1.Text = "PPLTOOLS";
&SubQryCrit1.Expr2Expression1 = &SubQryExpr1;

/* Second Criteria, which is for RECTYPE = 0 */
&MyCriteria2 = &MainQrySel.AddCriteria("SecondCriteria");

&MyCriteria2.Expr1Type = %Query_ExprField;
&MyCriteria2.AddExpr1Field(&MainRec.RecordAlias, "RECTYPE");

/* Since this is second criteria, we need a logical AND to state that - AND⇒
 RECTYPE = 0 */

&MyCriteria2.Logical = %Query_CombAnd;
```

```
&MyCriteria2.Operator = %Query_CondEqual;
&MyCriteria2.Expr2Type = %Query_ExprConstant;
&MyCrit2Expr2 = &MyCriteria2.AddExpr2Expression();
&MyCriteria2.Expr2Expression1 = &MyCrit2Expr2;
&MyCrit2Expr2.Text = "0";

&Rslt = &MyQuery.Save();

If &Rslt <> 0 Then
   /* save didn't complete */
   &COLL = &MySession.PSMessages;

      For &I = 1 to &COLL.Count
         &ERROR = &COLL.Item(&I);
         &TEXT = &ERROR.Text;
         /* do error processing */
      End-For;

      &COLL.DeleteAll();
   End-if;

Else

/* do error processing for not getting session */

End-if;
```

To add criteria to a query:

1. Get a session object.

   Before you can create a query, you have to get a session object. The session controls access to the query, provides error tracing, enables you to set the runtime environment, and so on. Then this program checks to verify that the session object is valid.

   ```
   &MySession = %Session;

   &MyQuery = &MySession.GetQuery();

   If &MySession <> Null Then
   ```

2. Access an existing query and get the main query select statement.

   Use the Open method to get the existing query. Then access the main select statement with the QuerySelect property.

   ```
   &MyQuery.Open("Table", False, True);

   &MainQrySel = &MyQuery.QuerySelect;
   ```

3. Add Query Record in the Main Select.

   Add the query record that you want to use.

```
&MainRec = &MainQrySel.AddQueryRecord("PSRECDEFN");
```

4. Add the displayed fields.

   You want to add the selected fields. Note instead of hardcoding the name of the record, this code example uses the Name property. Also, the code uses the RecordAlias property instead of hardcoding the alias. This makes the code easier to read, as well as easier to maintain. Specifying a column number also makes this an output field.

```
&QryFld = &MainQrySel.AddQuerySelectedField(&MainRec.Name, &MainRec.RecordAlias,⇒
 "RECNAME", "Record Name");
&QryFld.ColumnNumber = 1;
```

5. Specify the OrderBy value.

   Because we need to order by this field, the OrderByNumber of that field must be set also.

```
&QryFld.OrderByNumber = 1;
```

6. Add the first criteria.

   Add the first criteria. You don't want it added with any kind of operator, like an AND or an OR, so the **Logical** property of the first criteria is set with %Query_CombNotUsed. This is also used because it's the first non-having criteria of a query.

```
/* adding first criteria */
&Criteria1 = &MainQrySel.AddCriteria("FirstCriteria");

/* First criteria will not have any logical AND/OR */
   &Criteria1.Logical = %Query_CombNotUsed;
```

7. Add the first criteria field.

   The first field for the criteria is a QueryExpression type field. The type of the field *must* be set before the field is added.

```
   &Criteria1.Expr1Type = %Query_ExprField;
   &Criteria1.AddExpr1Field(&MainRec.RecordAlias, "RECNAME");
```

8. Add the condition for the first criteria and the subquery.

   The first criteria is the **WHERE RECNAME IN** portion of the SQL statement. The condition is considered 'in list', where the list is the result of the subquery. The expression is a subquery. Again, you have to set the type again before adding the subquery.

```
&Criteria1.Operator = %Query_CondInList;
&Criteria1.Expr2Type = %Query_ExprSubQuery;
&SubQrySel = &Criteria1.AddExpr2SubQuery();
```

9. Add the records for the subquery.

   Add the query record and the field from the query field, and make it an output field.

```
&SubQryRec = AddQueryRecord("PSPROJECTITEM");
&SubQryFld1 = &SubQrySel.AddQuerySelectedField(&SubQryRecName.Name, ⇒
&SubQryRec.RecordAlias, "OBJECTVALUE1", "Join Object");
&SubQryFld1.ColumnNumber = 1;
```

10. Add the criteria in the subquery.

    The following code adds the criteria for the subquery. Because this is the first non-having criteria in a select statement, the **Logical** property is set as %Query_CombNotUsed. Then the first expression is added as a field, and set to be equal to the second expression, PPLTOOLS. This is the **where PROJECTNAME = 'PPLTOOLS'** portion of the SQL statement.

```
/* Need criteria - PROJECTNAME = 'PPLTOOLS'  - in the subquery */
&SubQryCrit1 = &SubQrySel.AddCriteria("FirstSubQryCrit");

/* First criteria will not have any logical AND/OR */
   &SubQryCrit1.Logical = %Query_CombNotUsed;

   &SubQryCrit1.Expr1Type = %Query_ExprField;
   &SubQryCrit1.AddExpr1Field(&SubQryRec.RecordAlias, "PROJECTNAME");
&SubQryCrit1.Operator = %Query_CondEqual;

/* So that the criteria is constructed as - PROJECTNAME = 'PPLTOOLS'*/
&SubQryCrit1.Expr2Type = %Query_ExprConstant;
&SubQryExpr1 = &SubQryCrit1.AddExpr2Expression1();
&SubQryExpr1.Text = "PPLTOOLS";
   &SubQryCrit1.Expr2Expression1 = &SubQryExpr1;
```

11. Add the second criteria to the main select.

    Add the second criteria. Remember to set the type for the expression field first. Because this is the second criteria, we need a logical AND to state that this criteria is used with the first criteria.

```
/* Second Criteria, which is for RECTYPE = 0 */
&MyCriteria2 = &MainQrySel.AddCriteria("SecondCriteria");

&MyCriteria2.Expr1Type = %Query_ExprField;
&MyCriteria2.AddExpr1Field(&MainRec.RecordAlias, "RECTYPE");

/* Since this is second criteria, we need a logical AND to state that - AND⇒
 RECTYPE = 0 */

&MyCriteria2.Logical = %Query_CombAnd;

&MyCriteria2.Operator = %Query_CondEqual;
&MyCriteria2.Expr2Type = %Query_ExprConstant;

&MyCrit2Expr2 = &MyCriteria2.AddExpr2Expression();
&MyCriteria2.Expr2Expression1 = &MyCrit2Expr2;
&MyCrit2Expr2.Text = "0";
```

12. Save the data.

    When you execute the Save method, the new query is saved to the database.

```
&Rslt = &MyQuery.Save();

If &Rslt <> 0 Then
```

The **Save** method returns a numeric value: 0 if successful. You can use this value to do error checking.

13. Check Errors

    You can check if there were any errors using the PSMessages property on the session object.

```
   /* save didn't complete */
   &COLL = &MySession.PSMessages;
   For &I = 1 to &COLL.Count
      &ERROR = &COLL.Item(&I);
      &TEXT = &ERROR.Text;
      /* do error processing */
   End-For;
   &COLL.DeleteAll();
End-if;
```

    If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, you may want to delete it from the PSMessages collection.

CHAPTER 25

Record Class

This chapter provides an overview of Record class and discusses the following topics:

- Shortcut considerations.
- Record methods used to create SQL statements.
- Record object declaration.
- Scope of a record object.
- Record class built-in functions.
- Record class reference.

Understanding Record Class

A *record* object, instantiated from the Record class, is a single instance of a data within a row and is based on a record definition. A record object consists of one to *n* fields.

If a record object is instantiated using **GetRecord** (either the function or the method), the record object that is instantiated references the record from the current row in the data buffers, and its associated data.

If a record object is instantiated using the **CreateRecord** function, the record object that's instantiated is a freestanding record definition with its component set of field objects in the data buffer. The fields created by this function are initialized to null values, that is, they do *not* contain any data. Default processing is not performed. You can select into this record object using the **SelectByKey** method. You can also select into this record object using **SQLExec**.

CopyFieldsTo is a commonly used method for the record class. **Name**, **IsChanged**, and **FieldCount** are commonly used properties.

The record class is one of the data buffer access classes.

See Also

PeopleCode Developer's Guide, Data Buffer Access

Shortcut Considerations

An expression of the form

```
RECORD.recname.property
```

or

```
RECORD.recname.method(.  .  .)
```

is converted to an object expression by using **GetRecord**(**RECORD**.*recname*). For example, the following two lines of code are equivalent:

```
&MyRow = RECORD.EMPL_CHKLST_ITEM.ParentRow;

&MyRow = GetRecord(RECORD.EMPL_CHKLST_ITEM).ParentRow;
```

In addition, the default method for the Record class is the GetField method. This means you can access a field by just specifying the field name, instead of using GetField. For example, the following two lines of code, which both disable the EMPLID field, are equivalent:

```
&MyRec.EMPLID.Enabled = False;

&MyRec.GetField(FIELD.EMPLID).Enabled = False;
```

Note. If the field you're accessing has the same name as a record property (such as, Name) you can't use this method for accessing the field. You must use the GetField method.

Record Methods Used to Create SQL Statements

You can use the following record object methods to build and execute SQL statements:

- Delete
- SelectByKey
- Insert
- Update

The methods that result in a database update (specifically, UPDATE, INSERT, and DELETE) can only be issued in the following events:

- SavePreChange
- WorkFlow
- SavePostChange
- FieldChange

Remember that record UPDATEs, INSERTs, and DELETEs go directly to the database server, not to the Component Processor (although you can *view* data in the buffer using the PeopleCode debugger and other data buffer access classes). If a record method assumes that the database has been updated based on changes made in the component, that record method can be issued only in the SavePostChange event, because before SavePostChange none of the changes made to page data has actually been written back to the database.

If your application is repeating the same instruction many times, such as doing a million INSERTs, you should use the SQL object with the BulkMode property set to True, rather than the record SQL methods.

See Also

Chapter 25, Record Class , Delete, page 1447

Chapter 25, Record Class , SelectByKey, page 1454

Chapter 25, Record Class , Insert, page 1452

Chapter 25, Record Class , Update, page 1459

Chapter 31, SQL Class , page 1613

Data Type of a Record Object

Record objects are declared as type Record. For example,

```
Local Record &MYRECORD;
```

Scope of a Record Object

A record can only be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, Component Interface PeopleCode, and so on.

Record Class Built-in Functions

Chapter 1, PeopleCode Built-in Functions, CreateRecord , page 105

Chapter 1, PeopleCode Built-in Functions, GetRecord , page 265

Record Class Methods

In this section, we discuss each Record class method. The methods are discussed in alphabetical order.

CompareFields

Syntax

CompareFields(*recordobject*)

Description

The **CompareFields** method compares all like-named fields of the record object executing the method with the specified record object *recordobject*.

Parameters

Parameter	Description
recordobject	Specify a record object for use in the comparison. The specified record object does not have to refer to the same record as the record object executing the method.

Returns

A Boolean value; True if all like-named fields have the same value.

Example

```
&REC = GetRecord(RECORD.OP_METH_VW);
&REC2 = GetRecord(RECORD.OPC_METH);
If &REC2.CompareFields(&REC) Then
    WinMessage("All liked named fields have the same value");
End-If;
```

See Also

Chapter 25, Record Class , CopyChangedFieldsTo, page 1444, Chapter 25, Record Class , CopyFieldsTo, page 1445.

CopyChangedFieldsTo

Syntax

CopyChangedFieldsTo(*recordobject*)

Description

The **CopyChangedFieldsTo** method copies all like-named field values that have changed *from* the record object executing the method *to* the specified record object *recordobject.* This copies only changed field values. To copy *all*field values, use the CopyFieldsTo method.

Note. This method works only with database records. The Component Processor doesn't track the contents of work records, so there is no changed value to use for copying changed fields.

Parameters

Parameter	Description
recordobject	Specify a record object to be copied to. The specified record object does not have to refer to the same record as the record object executing the method.

Returns

None.

Example

```
Local Record &REC, &REC2;

&REC = GetRecord(RECORD.OPC_METH);

/* make changes to the values of fields in the record */

&REC2 = CreateRecord(RECORD.OPC_METH_WORK);

&REC.CopyChangedFieldsTo(&REC2);
```

See Also

Chapter 25, Record Class , CompareFields, page 1443, Chapter 25, Record Class , CopyFieldsTo, page 1445.

CopyFieldsTo

Syntax

CopyFieldsTo(*recordobject*)

Description

The **CopyFieldsTo** method copies all like-named field values *from* the record object executing the method *to* the specified record object *recordobject*. This copies all field values. To copy only changed field values, use the CopyChangedFieldsTo method.

Note. If you are copying to a derived work record, the IsChanged flag for the record is *not* set. Copying fields to a database record *does* set the IsChanged flag to True.

Parameters

Parameter	Description
recordobject	Specify a record object to be copied to. The specified record object does not have to refer to the same record as the record object executing the method.

Returns

None.

Example

```
Local Record &REC, &REC2;

&REC = GetRecord(RECORD.OPC_METH);
&REC2 = CreateRecord(RECORD.OPC_METH_WORK);
&REC.CopyFieldsTo(&REC2);
```

In the following example, records are copied into a record after being fetched.

```
Component number &displayNum;
Local SQL &sql;
Local Rowset &rs, &rs2;

&level0 = GetLevel0();
&displayNum = WS_NUM_ORDERS;

&rs = GetRowset(Record.WS_ORD_HDR_VW);
&rs.Flush();
WinMessage("1");
&sql = CreateSQL("%selectall(:1) where BUSINESS_UNIT=:2", Record.WS_ORD_HDR_VW,⇒
 "M04");
WinMessage("2");
&rec = CreateRecord(Record.WS_ORD_HDR_VW);

For &i = 1 To &displayNum

   If &sql.Fetch(&rec) Then

      &rs.InsertRow(&i);

      &rec.copyfieldsto(&rs.GetRow(&i).WS_ORD_HDR_VW);

      &rs2 = &rs.GetRow(&i).GetRowset(1);
      &rs2.Select(Record.WS_ORD_LINE_VW, "where BUSINESS_UNIT=:1 and ORDER_NO=:2",⇒
 &rec.BUSINESS_UNIT.value, &rec.ORDER_NO.value);

      /* Hide rows that do not contain Products */
      If &rs2 = Null Or None(&rs2.GetRow(1).WS_ORD_LINE_VW.⇒
ORDER_INT_LINE_NO.Value) Then
         For &j = 1 To &rs2.ActiveRowCount
            &rs2.GetRow(&j).Visible = False;
         End-For;
      End-If;
   End-If;
End-For;

&rs.GetRow(&i).Visible = False;
```

See Also

Chapter 25, Record Class , CompareFields, page 1443, Chapter 25, Record Class , CopyChangedFieldsTo, page 1444, Chapter 25, Record Class , IsChanged, page 1461.

DBPatternMatch

Syntax

DBPatternMatch(*Value*, *Pattern*, *CaseSensitive*)

Description

Use the **DBPatternMatch** function to match the string in *Value* to the given pattern.

You can use wildcard characters % and _ when searching. % means find all characters, while _ means find a single character. For example, to find if the string in *Value* started with the letter M, use "M%" for *Pattern*. To find either DATE or DATA, use "DAT_" for *Pattern*.

These characters can be escaped (that is, ignored) using a \. For example, to search for a value that contains the character %, use \% in *Pattern*.

If *Pattern* is an empty string, this function retrieves the value based only on the specified case sensitivity (that is, specifying "" for *Pattern* is the same as specifying "%").

Parameters

Parameter	Description
Value	Specify the string to be searched.
Pattern	Specify the pattern to be used when searching.
CaseSensitive	Specify whether the search is case-sensitive. This parameter takes a Boolean value: True, the search is case-sensitive, False, it is not.

Returns

A Boolean value: True if the string matches the pattern, False otherwise.

Delete

Syntax

```
Delete()
```

Description

The **Delete** method uses the key fields of the record and their values to build and execute a DELETE SQL statement which deletes the record (row of data) from the SQL data table.

This method, like the DeleteRow Rowset class method, initially marks the record or row as needing to be deleted. At save time the row is actually deleted from the database and cleared from the buffer.

Because this method results in a database change, it can be issued only in the following events:

- SavePreChange
- WorkFlow
- SavePostChange

If your application is repeating the same instruction multiple times, such as doing a million DELETEs, use the SQL object with the BulkMode property set to True, rather than the record SQL methods.

For every record deleted by the **Delete** method, if the set language is not the base language and the record has related language records, the **Delete** method tries to do related language processing.

Parameters

None.

Returns

The result is True on successful completion, False if the record was not found. Any other conditions cause termination.

Example

Suppose that KEYF1 and KEYF2 are the two key fields of record definition MYRECORD. The following code deletes the database record that has KEYF1 equal to "A" and KEYF2 equal to "X":

```
Local record &REC;
&REC = CreateRecord(RECORD.MYRECORD);
&REC.KEYF1.Value = "A";
&REC.KEYF2.Value = "B";
&REC.MYRF3.Value = "X";
&REC.MYRF4.Value = "Y";
&REC.Insert();
```

See Also

Chapter 25, Record Class , SelectByKey, page 1454, Chapter 25, Record Class , Insert, page 1452, Chapter 25, Record Class , Update, page 1459, Chapter 27, Rowset Class , DeleteRow, page 1484, Chapter 31, SQL Class , page 1613, *PeopleTools Global Technology*, Using Related Language Tables.

ExecuteEdits

Syntax

ExecuteEdits([*editlevel*]);

where *editlevels* is a list of values in the form:
editlevel1 [+ *editlevel2*] . . .]);

and where *editleveln* is one of the following constants:

%Edit_DateRange

%Edit_OneZero

%Edit_PromptTable

%Edit_Required

%Edit_TranslateTable

%Edit_YesNo

Description

The **ExecuteEdits** method executes the standard system edits on every field in the record. The types of edits performed depends on the *editlevel*. If no *editlevel* is specified, all system edits are executed. All *editlevels* are already defined for the record definition or for the field definition, that is:

- Reasonable Date Range (Is the date contained within the specified reasonable date range?)
- 1/0 (Do all 1/0 fields contain only a 1 or 0?)
- Prompt Table (Is field data contained in the specified prompt table?)
- Required Field (Do all required fields contain data? For numeric or signed fields, checks for NULL or 0 values.)
- Translate Table (Is field data contained in the specified translate table?)
- Yes/No (Do all yes/no fields only contain only yes or no data?)

If any of the edits fail, the status of the property **IsEditError** is set to False for the record. The field property **EditError** is set to True for any fields that are in error. In addition, the Field class properties **MessageNumber** and **MessageSetNumber** are set to the number of the returned message and message set number of the returned message, for each field in error.

You *must* use the SetEditTable method to set the prompt tables for fields that are defined with %EditTable in the record definition.

If you're running an Application Engine program, and you want to do set based ExecuteEdits (as opposed to row-by-row) consider using the meta-SQL construct %ExecuteEdits.

See Chapter 25, Record Class , SetEditTable, page 1457; Chapter 12, Field Class , EditError, page 854; Chapter 12, Field Class , MessageNumber, page 863; Chapter 12, Field Class , MessageSetNumber, page 863 and *PeopleSoft Application Engine*, Using Meta-SQL and PeopleCode, %ExecuteEdits.

Considerations for ExecuteEdits and SetEditTable

If an effective date is a key on the prompt table, and the record being edited doesn't contain an EFFDT field, the current datetime is used as the key value.

If a SETID is a key on the prompt table, and the record being edited doesn't contain a SETID field, the system looks for SETID on the other records in the current row first, then searches parent rows.

For *all other keys*, the value used to "select" the correct row in the prompt table record comes only from the *record* executing the method. No other records (or key field values) are used. You may get unexpected results if not all the keys for the prompt table are filled in (or filled in correctly.)

Considerations for ExecuteEdits and Numeric Fields

A 0 (zero) may or may not be a valid value for a numeric field. ExecuteEdits handles numeric fields in different ways, depending on whether the field is required and a prompt table is used:

- If the numeric field is required and there is no prompt edit: 0 is considered valid, so numeric fields never fail the required test.
- If the numeric field is required and there is a prompt edit: Prompt edit is run and the value must be valid.
- If the numeric field is not required and there is a prompt edit: 0 is considered valid, so the prompt edit is skipped.

Parameters

Parameter	Description
editlevel	Specifies the standard system edits to be performed against every field on every record. If *editlevel* isn't specified, all system edits are performed. *editlevel* can be any of the following system variables. • %Edit_DateRange • %Edit_OneZero • %Edit_PromptTable • %Edit_Required • %Edit_TranslateTable • %Edit_YesNo

Returns

None.

Example

The following is an example of a call to execute Required Field and Prompt Table edits:

```
&REC.ExecuteEdits(%Edit_Required + %Edit_PromptTable);
```

The following is an example showing how ExecuteEdits() could be used:

```
&REC.ExecuteEdits();
If &REC.IsEditError Then
 LogError();  /*application specific call */
End-If;
```

See Also

Chapter 25, Record Class , SetEditTable, page 1457, Chapter 25, Record Class , IsEditError, page 1462, Chapter 12, Field Class , EditError, page 854, Chapter 12, Field Class , MessageNumber, page 863, Chapter 12, Field Class , MessageSetNumber, page 863.

GetField

Syntax

GetField({*n* | FIELD.*fieldname*})

Description

The **GetField** method instantiates a field object for the specified field associated with the record. This is the *default method* for the record object. This means that any record object, followed by a parameter list, acts as if GetField is specified.

Note. If the field you're accessing has the same name as a record property (that is, Name or FieldCount) you can't use the default method for accessing the field. You must specify **GetField**.

For example, the following is invalid for accessing the value of a field called NAME:

```
&NUMBER = &REC.NAME.Value;
```

You must use the following code to get the value of a field called NAME:

```
&NUMBER= &REC.GetField(FIELD.NAME).Value;
```

Parameters

Parameter	Description
n | **FIELD**.*fieldname*	Specify a field to be used for instantiating the field object. You can specify either *n* or FIELD.*fieldname*, Specifying *n* creates a field object for the *nth* field in the record. This might be used if writing code that needs to examine all fields in a record without being aware of the name. Specifying FIELD.*fieldname* creates a field object for the field *fieldname*.

Note. There is no way to predict the order the fields will be accessed. Use the *n* option only if the order in which the fields are processed doesn't matter.

Returns

A field object.

Example

```
&REC.GetField(FIELD.CHARACTER).Value = "Hello";
```

As GetField is the default method for a record object, the following code is identical to the previous code:

```
&REC.CHARACTER.VALUE = "Hello";
```

The following code creates an array containing all the names of all the fields in a related language record.

```
Local array of string &FIELD_LIST_ARRAY;
&FIELD_LIST_ARRAY = CreateArray();
For &I = 1 to &REC_RELATED_LANG.FieldCount
 &FIELD_LIST_ARRAY.Push(&REC_RELATED_LANG.GetField(&I).Name);
End-For;
```

The GetField method requires either FIELD.fieldname or a number. It won't take a string field name. However, you can use the @ operator to convert the string field name into a **FIELD**.*fieldname* reference, as follows:

```
&REC = GetRecord();
&REC2 = GetLevel0(1).EMPL_CHECKLIST;
&FIELD2 = &REC.GetField(@ "FIELD." | &REC2.getfield(&I).Name);
```

The following code converts field name strings (using the @ symbol) to component names to get the value of all the fields in a subrecord. Note the code in **Keyword style**.

```
Function get_draft_dst_codes
   /* Load dst id codes into record structures */
   &DSTCODES = CreateRecord(RECORD.DR_DST_CODE_SBR);
```

```
   &DRAFTITEM = GetRecord(RECORD.DRAFT_ITEM);
   &DRAFTITEM.CopyFieldsTo(&DSTCODES);
   /* If any missing get from AR dist code, then Draft Type-BU, then BU */
   For &I = 1 To &DSTCODES.FieldCount
      If None(&DSTCODES.GetField(&I).Value) Then
         get_ar_dst_code();
         &NAME = &DSTCODES.GetField(&I).Name;
         If All(&DST.GetField(@ ("FIELD." | &NAME)).Value) Then
            &DSTCODES.GetField(&I).Value = &DST.GetField(@("FIELD." |⇒
&NAME)).Value;
         Else
            If All(&R_DRAFTBU.GetField(@("FIELD." | &NAME)).Value) Then
               &DSTCODES.GetField(&I).Value = &R_DRAFTBU.GetField(@("FIELD." |⇒
&NAME)).Value;
            Else
               &DSTCODES.GetField(&I).Value = &R_ARBU.GetField(@("FIELD." |⇒
&NAME)).Value;
            End-If;
         End-If;
      End-If;
   End-For;
   /* Copy the defaulted values back to draft item */
   &DSTCODES.CopyFieldsTo(&DRAFTITEM);
End-Function;
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetField , page 242.

Insert

Syntax

Insert()

Description

The **Insert** method uses the field names of the record and their values to build and execute an Insert SQL statement which adds the given record (row of data) to the SQL table.

Because this method results in a database change, it can only be issued in the following events:

- SavePreChange
- WorkFlow
- SavePostChange

If your application is repeating the same instruction many times, such as doing a million INSERTs, use the SQL object with the BulkMode property set to True, rather than the record SQL methods.

If you're using a record created using CreateRecord, all fields are initially set to "", 0, or NULL, depending on the type of field. If you don't specify values for these fields, these initial values are written to the database when the insert is executed. If you want the default values for the fields inserted instead, use the SetDefault method prior to using the **Insert** method.

For every record inserted with the **Insert** method, if the set language is not the base language and the record has related language records, the **Insert** method tries to do related language processing.

See Also

Chapter 31, SQL Class , page 1613, *PeopleTools Global Technology*, Using Related Language Tables, Chapter 25, Record Class , SetDefault, page 1456.

Parameters

None.

Returns

The result is True on successful completion, False if there was a record with those keys already in the database, that is, if a duplicate record was found. Any other conditions cause termination.

Example

Suppose that KEYF1 and KEYF2 are the key fields of record definition MYRECORD, and that this record definition also contains the field definitions MYRF3, MYRF4. The following code adds a MYRECORD record (row of data) to the database, with KEYF1 set to "A", KEYF2 set to "B", MYRF3 set to "X", and MYRF4 set to "Y":

```
Local record &REC;
&REC = CreateRecord(RECORD.MYRECORD);
&REC.KEYF1.Value = "A";
&REC.KEYF2.Value = "B";
&REC.MYRF3.Value = "X";
&REC.MYRF4.Value = "Y";
&REC.Insert();
```

See Also

Chapter 25, Record Class , Delete, page 1447, Chapter 25, Record Class , SelectByKey, page 1454, Chapter 25, Record Class , Update, page 1459.

SearchClear

Syntax

SearchClear()

Description

The **SearchClear** method clears the field values for all search keys for a record.

Considerations Using SearchClear and SearchDefault

The field property SearchDefault sets a field to its default value (if there is one) immediately after SearchInit PeopleCode finishes. SearchDefault overrides SearchClear. If you call SearchClear for a record, then use SearchDefault for a field, the field is set to its default value and the search key values for the rest of the record are cleared.

Parameters

None.

Returns

None.

Example

```
Local Record &REC;

&REC = GetRecord();
&REC.SearchClear();
```

See Also

Chapter 12, Field Class , SearchDefault, page 865, Chapter 12, Field Class , SearchClear, page 850.

SelectByKey

Syntax

SelectByKey()

Description

The **SelectByKey** method uses the key field names of the record and their values to build and execute a Select SQL statement. The field values are then fetched from the database SQL table into the record object executing the method, and the Select statement is closed.

Note. You can't use this method in dynamic views.

If you don't specify all the key fields for a record, those you exclude are added to the WHERE clause with the condition equal to a blank value. If not all keys are set and more than one row is retrieved, you won't receive an error and **SelectByKey** won't fetch any data.

SelectByKey returns a single row of data. To fetch more than one row, use the SQL object.

For every record read by the **SelectByKey** method, if the set language is not the base language and the record has related language records, the **SelectByKey** method tries to do related language processing.

You can also select into a record using SQLExec and meta-SQL. For example, if you need to use SelectByKey with an effective date, you can use the meta-SQL %SelectByKeyEffDt in a SQLExec statement. In the following example, &RECOBJ is the name of a record created using the **CreateRecord** function.

```
SQLExec("%SelectByKeyEffDt(:1, :2)", &RECOBJ, %Date, &RECOBJ);
```

See Also

Chapter 31, SQL Class , page 1613, *PeopleTools Global Technology*, Using Related Language Tables, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442.

Parameters

None.

Returns

The result is True on successful completion, False if the record was not found. Any other conditions cause termination.

Example

Suppose that KEYF1 and KEYF2 are the two key fields of record definition MYRECORD. The following code reads the database record that has KEYF1 equal to "A" and KEYF2 equal to "X" into the &REC record object:

```
Local record &REC;
&REC = CreateRecord(RECORD.MYRECORD);
&REC.KEYF1.Value = "A";
&REC.KEYF2.Value = "X";
&REC.SelectByKey();
```

See Also

Chapter 25, Record Class , Delete, page 1447, Chapter 25, Record Class , Insert, page 1452, Chapter 25, Record Class , Update, page 1459.

SelectByKeyEffDt

Syntax

SelectByKeyEffDt(*Date*)

Description

Use the **SelectKeyByEffDt** to build and execute a SQL Select statement to get the current effective row based on an "as of date". The field values are then fetched from the database SQL table into the record object executing the method, and the Select statement is closed.

Note. You can’t use this method in dynamic views.

SelectByKeyEffDt returns a single row of data. To fetch more than one row, use the SQL object.

For every record read by the **SelectByKeyEffDt** method, if the set language is not the base language and the record has related language records, the **SelectByKeyEffDt** method tries to do related language processing. The system assumes that both the base table and the related language table are effective-dated and that the effective-date keys are the same in each.

To use other keys, more than just the effective date, use the SelectByKey Record class method.

Parameters

Parameter	Description
Date	Specify the "as of" date you want to use as the effective date for the Select statement.

Returns

The result is True on successful completion, False if the record was not found. Any other conditions cause termination.

Example

```
Local Record &Rec;
Local any &Date;

&Date = %Date;
&Rec = CreateRecord(Record.COMPANY_TBL);
&Rec.COMPANY.Value = "CCB";

If &Rec.SelectByKeyEffDt(&Date) Then
   &Out = &Rec.DESCR.Value | " as of " | &Rec.EFFDT.Value | "->" |⇒
 &Rec.STREET1.Value;
   WinMessage("Found this company: " | &Out);
Else
   WinMessage("Not there!");
End-If;
```

See Also

Chapter 25, Record Class , Delete, page 1447, Chapter 25, Record Class , Insert, page 1452, Chapter 25, Record Class , Update, page 1459, Chapter 25, Record Class , SelectByKey, page 1454, Chapter 31, SQL Class , page 1613, *PeopleTools Global Technology*, Using Related Language Tables.

SetDefault

Syntax

SetDefault()

Description

SetDefault sets the value of every field in the record to a null value, or to a default, depending on the type of field.

- If this method is used against data from the Component buffers, the next time default processing occurs, it is set to its default value: either a default specified in its record field definition or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.
- If this method is used with a field that isn't part of the data buffer (for example, a field in a record object instantiated with CreateRecord) the field is automatically set to its default value if one is set for the *field*, not for the record field. Any FieldDefault PeopleCode does *not* run on these types of fields. To set the default values for just one field, use the **SetDefault** field class method.

Blank numbers correspond to zero on the database. Blank characters correspond to a space on the database. Blank dates and long characters correspond to NULL on the database. **SetDefault** gives each field data type its proper value.

Parameters

None.

Returns

None.

Example

```
&CHARACTER.SetDefault();
```

See Also

Chapter 12, Field Class , SetDefault, page 851, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Default Processing.

SetEditTable

Syntax

SetEditTable(*%PromptField*, RECORD.*recordname*)

Description

The **SetEditTable** method works with the ExecuteEdits method. It sets the value of a field on a record that has its prompt table defined as %*PromptField* value. %PromptField values are used to dynamically change the prompt record for a field.

There are several steps to setting up a field so that you can dynamically change its prompt table.

To set up a field with a dynamic prompt table:

1. Define a field in the DERIVED record called *fieldname*.
2. In the record definition for the field you want to have a dynamic prompt table, define the prompt table for the field as %*PromptField*, where *PromptField* is the name of the field you created in the DERIVED record.

   Note. When you use **SetEditTable**, you don't have to add a hidden field to the page.

3. Use **SetEditTable** to dynamically set the prompt table.

%PromptField is the name of the field on the DERIVED work record. RECORD.*recordname* is the name of the record to be used as the prompt table.

```
&REC.SetEditTable("%EDITTABLE1", Record.DEPARTMENT);
&REC.SetEditTable("%EDITTABLE2", Record.JOB_ID);
&REC.SetEditTable("%EDITTABLE3", Record.EMPL_DATA);
&REC.ExecuteEdits();
```

Every field on a record that has the prompt table field %EDITTABLE1 will have the same prompt table, that is, DEPARTMENT.

See Chapter 25, Record Class , ExecuteEdits, page 1448 and *PeopleSoft Application Designer*, Creating Record Definitions.

Considerations for SetEditTable and ExecuteEdits

The keys used to "select" the correct row in the prompt table record come only from the record object executing the method. All the keys from the component are not used. You may get unexpected results if not all the keys for the prompt table are filled in (or filled in correctly.)

Parameters

Parameter	Description
%PromptField	Specifies the name of the field in the DERIVED record that has been defined as the prompt table for a field in the record definition.
RECORD.*recordname*	Specifies the name of the record definition that contains the correct standard system edits to be performed for that field in the record.

Returns

None.

Example

```
/* To Set the Prompt Table */
&Der_EditRec = &TSKPRF_Det_Row.GetRecord(Record.DERIVED);
&Prompt_Editname = &Der_EditRec.EDITTABLE.Value;
&Prompt_Edittable = "Record." | &Prompt_Editname;
&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE", @&Prompt_Edittable);

&Prompt_Editname2 = &Der_EditRec.EDITTABLE2.Value;
&Prompt_Edittable2 = "Record." | &Prompt_Editname2;
&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE2", @&Prompt_Edittable2);

&Prompt_Editname3 = &Der_EditRec.EDITTABLE3.Value;
&Prompt_Edittable3 = "Record." | &Prompt_Editname3;
&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE3", @&Prompt_Edittable3);

&Prompt_Editname4 = &Der_EditRec.EDITTABLE4.Value;
&Prompt_Edittable4 = "Record." | &Prompt_Editname4;
&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE4", @&Prompt_Edittable4);

&Prompt_Editname5 = &Der_EditRec.EDITTABLE5.Value;
&Prompt_Edittable5 = "Record." | &Prompt_Editname5;
&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE5", @&Prompt_Edittable5);

&Prompt_Editname6 = &Der_EditRec.EDITTABLE6.Value;
&Prompt_Edittable6 = "Record." | &Prompt_Editname6;
&TSKPRF_Det_Rec.SetEditTable("%EDITTABLE6", @&Prompt_Edittable6);
```

```
&TSKPRF_Det_Rec.ExecuteEdits(%Edit_Required + %Edit_PromptTable);
```

See Also

Chapter 25, Record Class , ExecuteEdits, page 1448, Chapter 25, Record Class , IsEditError, page 1462, Chapter 12, Field Class , EditError, page 854, Chapter 12, Field Class , MessageNumber, page 863, Chapter 12, Field Class , MessageSetNumber, page 863.

Update

Syntax

Update([*KeyRecord*])

Description

The **Update** method uses the changed fields of the record and their values to build and execute an Update SQL statement, updating the SQL table. If you are updating the key fields of the record, you must supply a record object *KeyRecord* that contains the old values of the key record fields.

Because this method results in a database change, it can only be issued in the following events:

- SavePreChange
- WorkFlow
- SavePostChange

If your application is repeating the same instruction many times, such as doing a million UPDATEs, use the SQL object with the BulkMode property set to True, rather than the record SQL methods.

For every record updated by the **Update** method, if the set language is not the base language and the record has related language records, the **Update** method tries to do related language processing.

Note. This method does *not* update fields marked as “system maintained” in Application Designer.

See Also

Chapter 31, SQL Class , page 1613, *PeopleTools Global Technology*, Using Related Language Tables.

Parameters

Parameter	Description
KeyRecord	If you’re updating the key fields of the record, you must supply the old key field values in the record object *KeyRecord*.

Returns

The result is True on successful completion, False if the record was not found or (when updating the key fields) a duplicate record was found. Any other conditions cause termination.

Example

Suppose that KEYF1 and KEYF2 are the key fields of record definition MYRECORD, and that this record definition also contains the record field definitions MYRF3, MYRF4. The following code updates the MYRECORD record in the database with KEYF1 set to "A", KEYF2 set to "B", setting MYRF3 to "X" and MYRF4 to "Y":

```
Local record &REC;
&REC = CreateRecord(RECORD.MYRECORD);
&REC.KEYF1.Value = "A";
&REC.KEYF2.Value = "B";
&REC.MYRF3.Value = "X";
&REC.MYRF4.Value = "Y";
&REC.Update();
```

This code updates the MYRECORD record in the database with KEYF1 of "A" and KEYF2 of "B", setting KEYF2 to "C", MYRF3 to "M" and MYRF4 to "N":

```
Local record &REC1, &REC2;
&REC1 = CreateRecord(RECORD.MYRECORD);
&REC2 = CreateRecord(RECORD.MYRECORD);
&REC1.KEYF1.Value = "A";
&REC1.KEYF2.Value = "B";
&REC2.KEYF1.Value = "A";
&REC2.KEYF2.Value = "C";
&REC2.MYRF3.Value = "M";
&REC2.MYRF4.Value = "N";
&REC2.Update(&REC1);
```

See Also

Chapter 25, Record Class , Delete, page 1447, Chapter 25, Record Class , SelectByKey, page 1454, Chapter 25, Record Class , Insert, page 1452.

Record Class Properties

In this section, we discuss the Record class properties. The properties are listed in alphabetical order.

FieldCount

Description

This property returns the total number of fields contained in the record. This value is a number.

This property is read-only.

Example

```
WinMessage("This record has this many fields : " | &REC.FieldCount);
```

fieldname

Description

If a field name is used as a property, it accesses the field object with that name. This means the following code:

```
&REC.fieldname
```

acts the same as

```
&REC.GetField(FIELD.fieldname);
```

Note. If the field you're accessing has the same name as a record property (such as, Name) you can't use this method for accessing the field. You must use the GetField method.

If you combine the *fieldname* property with the *recname* property, you obtain the specified field object associated with that record from the row.

```
&ROW.recname.fieldname
```

This property is read-only.

Example

```
&REC.CHARACTER.Enabled = True;
```

IsChanged

Description

This property returns True if any field value on the primary database record of the row has been changed.

Note. This property is for use only with the primary database record. It does not return valid results if used with a work record.

This property is read-only.

Considerations Using IsChanged

This property and the IsChanged row property do *not* always return identical values.

If a row in a scroll contains multiple records (such as a primary database record and one or more work records), the IsChanged row property returns True if *any* of the records in the row are changed. The IsChanged record property returns True *only* if a specific record, namely, the primary database record, is changed.

If a row is deleted from a scroll, *only* the primary database record has its IsChanged property marked to False (since the row has been deleted.) Any work records in the row *still* have their IsChanged properties set to True.

Example

```
If &REC.IsChanged Then
    Warning("This Record has been changed");
End-if;
```

IsDeleted

Description

This property is True if the record has been deleted. For a level 0 record, it is possible for a record to be deleted without the whole row being deleted. For other levels, this property is the same as the row property IsDeleted.

This property is read-only.

Example

```
&tmp = &REC.IsDeleted;
```

See Also

Chapter 26, Row Class , IsDeleted, page 1473.

IsEditError

Description

This property is True if an error has been found for any field associated with the record after executing the **ExecuteEdits** method. This property can be used with the Field class properties **EditError** (to find out which field is in error), and **MessageSetNumber** and **MessageNumber** to find the error message set number and error message number.

This property is read-only.

Example

The following is an example showing how **IsEditError**, along with the **ExecuteEdits** method could be used:

```
&REC.ExecuteEdits();
If &REC.IsEditError Then
 For &I = 1 to &REC.FieldCount
  If &REC.GetField(&I).EditError Then
   LOG_ERROR();  /* application specific call */
  End-If;
 End-For;
End-If;
```

See Also

Chapter 25, Record Class , ExecuteEdits, page 1448, Chapter 25, Record Class , IsEditError, page 1462, Chapter 12, Field Class , EditError, page 854, Chapter 12, Field Class , MessageNumber, page 863, Chapter 12, Field Class , MessageSetNumber, page 863.

Name

Description

This property returns the name of the record definition of the record as a string.

To access an SQL table name for a record, use the %Table meta-SQL construct.

This property is read-only.

Example

```
WinMessage("The name of this record is : " | &REC.Name);
```

See Also

Chapter 2, Meta-SQL , %Table , page 551.

ParentRow

Description

This property returns the row object for the row containing the record. If the record object was created with the CreateRecord built-in function, the value for **ParentRow** is null because the record object isn't part of a row.

This property is read-only.

Example

```
&ROWOBJECT = &REC.ParentRow;
    &TMP = "The row number of the parent row is " | &ROWOBJECT.RowNumber;
/* note that RowNumber is a property of the row class  */
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateRecord , page 105.

RelLangRecName

Description

This property returns the name of the related language record as a string. The value will be a null string ("") if there is no related language record.

Example

```
Local Record &REC;

&REC = GetRecord();
&RECNAME_BASE = &REC.Name;
&RELLANGRECNAME = &REC.RelLangRecName;
SQLExec("select RELLANGRECNAME from PSRECDEFN where RECNAME = :1", &RECNAME_BASE,⇒
 &RELLANGRECNAME);
```

CHAPTER 26

Row Class

This chapter provides an overview of Row class and discusses the following topics:

- Shortcut considerations.
- Data type of a row object.
- Scope of a row object.
- Row class built-in function.
- Row class methods.
- Row class properties.

Understanding Row Class

A *row* object, instantiated from the Row class, is a single row of data that consists of one to *n* records of data. A single row in a component scroll is a row.

A row object is always associated with a rowset object, that is, you cannot instantiate a row object without first either explicitly or implicitly instantiating a rowset object.

A row may have one to *n* child rowsets. For example, a row in a level two scroll may have *n* level three child rowsets.

CopyTo and **GetRecord** are two commonly use methods for this class. **Visible** and **IsChanged** are two commonly used properties for this class.

The row class is one of the data buffer access classes.

If a rowset object is instantiated using the **CreateRowset** function, the rowset object that's instantiated is a standalone rowset. Delete and insert activity on these types of rowsets aren't automatically applied at save time. Use standalone rowsets for work records.

See Also

PeopleCode Developer's Guide, Data Buffer Access

PeopleCode Developer's Guide, Using Methods and Built-In Functions, Using Standalone Rowsets

Shortcut Considerations

The default method for the row class is GetRecord. This means you can specify just a record name to access a record on the row. For example, the following two lines of code are equivalent:

```
&Count = &MyRow.EMPL_CHECKLIST.FieldCount;

&Count = &MyRow.GetRecord(RECORD.EMPL_CHECKLIST).FieldCount;
```

In addition, the row class has a method *scrollname*. This enables you to access a specific child rowset *and* row within that rowset. This isn't a default method: you're not accessing a child object, but rather, something within that object. For example, the following two lines of code are equivalent:

```
&ChildRow = &MyRow.EMPL_CHKLST_ITM(&I);

&ChildRow = &MyRow.GetRowset(SCROLL.EMPL_CHKLST_ITM).GetRow(&I);
```

Data Type of a Row Object

Row objects are declared as type Row. For example,

```
Local Row &MYROW;
```

Scope of a Row Object

A Row can only be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, Component Interface PeopleCode, and so on.

Row Class Built-in Function

Chapter 1, PeopleCode Built-in Functions, GetRow , page 267

Row Class Methods

In this section, we discuss each Row class method. The methods are discussed in alphabetical order.

CopyTo

Syntax

CopyTo(*row*)

Description

The **CopyTo** method copies *from*the row executing the method *to*the specified target row, copying *like-named* record fields and subscrolls at corresponding levels. The target row object must be from a "related" rowset, that is, built from the same records, but it can be under a different parent at higher levels.

This method copies *all* the records from the row object executing the method to the target row, not just the primary data record associated with the rowset.

Parameters

Parameter	Description
row	Specify a target row. This must be a row object, not a row number.

Returns

None

Example

The following example copies rows from a child rowset (DERIVED_HR) to the current rows.

```
Local Row &ROW;
Local Rowset &RS1;

&RS1 = GetRowset();
For &I = 1 to &RS1.ActiveRowCount
  &ROW = GetRowset(SCROLL.DERIVED_HR).GetRow(&I);
  &ROW.CopyTo(GetRow(&I));
End-For;
```

See Also

Chapter 26, Row Class , ParentRowset, page 1475, Chapter 25, Record Class , CopyFieldsTo, page 1445, Chapter 25, Record Class , CopyChangedFieldsTo, page 1444.

GetNextEffRow

Syntax

GetNextEffRow()

Description

The **GetNextEffRow** method finds the row, in the same rowset as the row executing the method, that has the next effective date (when compared to the row executing the method.)

Parameters

None

Returns

A row object. If there is no row with a later effective date, this method returns a null object.

Example

```
&ROW2 = &ROW.GetNextEff();

If &ROW2 <> NULL Then
  &TMP = &ROW2.RowNumber;
  /* other processing */
Else
  /* no effective dated row - do error processing */
End-if;
```

See Also

Chapter 26, Row Class , GetPriorEffRow, page 1468, Chapter 27, Rowset Class , GetCurrEffRow, page 1491, Chapter 27, Rowset Class , DeleteEnabled, page 1504.

GetPriorEffRow

Syntax

```
GetPriorEffRow()
```

Description

The **GetPriorEffRow** method finds the row, from the same rowset as the row executing the method, that has the prior effective date to the row executing the method.

Parameters

None

Returns

A row object. If there is no row with a prior effective date, this method returns a null object.

Example

```
&TMP = &ROW.GetPriorEffrow().RowNumber;
```

See Also

Chapter 26, Row Class , GetNextEffRow, page 1467, Chapter 27, Rowset Class , GetCurrEffRow, page 1491, Chapter 27, Rowset Class , DeleteEnabled, page 1504.

GetRecord

Syntax

GetRecord({*n* | RECORD.*recname*})

Description

The **GetRecord** method creates a record object that references the specified record within the current row object. This is the *default method* for a row object. This means that any row object, followed by a parameter list, acts as if GetRecord is specified.

Parameters

Parameter	Description
n | RECORD.*recname*	Specify a record to be used for instantiating the record object. You can specify either *n* or RECORD.*recname*, Specifying *n* creates a record object for the *nth* record in the row. This might be used if writing code that needs to examine all records in a row without being aware of the record name. Specifying RECORD.*recname* creates a record object for the record *recname*.

Note. There is no way to predict the order the records will be accessed. Use the *n* option only if the order in which the records are processed doesn't matter.

Returns

A record object.

Example

The following example gets a record, then assigns the name of the record to the variable &TMP:

```
&REC = &ROW.GetRecord(RECORD.QEPC_LEVEL1_REC);

&TMP = &REC.NAME;  /* note that NAME is a property of the record class */
```

Because GetRecord is the default method for a row, the following code is identical to the previous:

```
&REC = &ROW.QEPC_LEVEL1_REC;

&TMP = &REC.NAME;
```

The following example loops through all the records for a row:

```
Local rowset &LEVEL1;
Local row &ROW;
Local record &REC;

&LEVEL1 = GetRowset(SCROLL.EMPL_CHECKLIST);
For &I = 1 to &LEVEL1.ActiveRowCount
  &ROW = &LEVEL1.GetRow(&I);
  For &J = 1 to &ROW.RecordCount
   &REC = &ROW.GetRecord(&J);
   /* do processing */
```

```
    End-For;
End-For;
```

The following function takes a rowset and a record, passed in from another program. GetRecord won't take a variable for the record, however, using the @ symbol you can convert the string to a component record name.

```
Function Get_My_Row(&PASSED_ROWSET, &PASSED_RECORD)

   For &ROWSET_ROW = 1 To &PASSED_ROWSET.RowCount
      &UNDERLYINGREC = "RECORD." | &PASSED_ROWSET.DBRecordName;
      &ROW_RECORD = &PASSED_ROWSET.GetRow(&ROWSET_ROW).GetRecord(@&UNDERLYINGREC);

      /* Do other processing */

   End-For;

End-Function;
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetRecord , page 265, Chapter 1, PeopleCode Built-in Functions, CreateRecord , page 105.

GetRowset

Syntax

GetRowset({*n* | SCROLL.*scrollname*})

Description

The **GetRowset** method creates a rowset object that references a child rowset of the current row. *scrollname* must specify the primary record for the child rowset.

Parameters

Parameter	Description
n | **SCROLL**.*scrollname*	Specify a rowset to be used for instantiating the rowset object. You can specify either *n* or SCROLL.*scrollname*. Specifying *n* creates a rowset object for the *nth* child rowset in the row. This might be used if you are writing code that examines all rowsets in a row without being aware of the name. Specifying SCROLL.*scrollname* creates a rowset object for the record *scrollname*. *scrollname* must specify the primary record for the child rowset.

Note. There is no way to predict the order the rowsets will be accessed. Use the *n* option only if the order in which the rowsets are processed doesn't matter.

Returns

A rowset object.

Example

```
&CHILDROWSET = &ROW.GetRowset(SCROLL.QEPC_LEVEL1_REC);
```

See Also

Chapter 26, Row Class , scrollname, page 1471, Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268.

scrollname

Syntax

scrollname(*n*)

Description

If *scrollname* is used as a method name for a row, it is used to select a child rowset and a row within that rowset.

```
&ROW.scrollname(n)
```

acts the same as

```
&ROW.GetRowset(SCROLL.scrollname).GetRow(n)
```

Parameters

Parameter	Description
n	Must be a valid row number for the selected child rowset.

Returns

A row object.

Example

```
&SUBROW1 = &ROW.QEPC_LEVEL1_REC(1);
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268.

Row Class Properties

In this section, we discuss the Row class properties. The properties are discussed in alphabetical order.

ChildCount

Description

This property returns the number of child rowsets of the row. It is defined by the "structure" of the scroll, so it is the same for all rows of the rowset.

It might be used, in conjunction with the **GetRowset** method, to write code that examines all child rowsets.

This property is read-only.

Example

```
For &I = 1 to &ROW.ChildCount
  &ROWSET = &ROW.GetRowset(&I);
  /* do processing */
End-For;
```

DeleteEnabled

Description

This property determines whether a row can be deleted (the equivalent of the user pressing ALT+8 and ENTER). This property takes a Boolean value.

Note. This property controls only whether an end-user can delete a row. Rows can still be deleted using PeopleCode.

This property is typically used to prevent deletion of an individual row that the other properties controlling deletion would allow to be deleted. The following table goes through the Boolean logic.

Will delete be allowed by user?	Others enable	Others disable
DeleteEnabled = True	Yes	No
DeleteEnabled = False	No	No

The initial value of this property is True.

Note. If No Row Delete is selected in PeopleSoft Application Designer, setting the DeleteEnabled row property to True will *not* override this value. If you want to control whether a row can be deleted at runtime, you should *not* select No Row Delete at design time. Likewise, if the DeleteEnabled **rowset** property is False, setting the DeleteEnabled **row** property to True will *not* override the rowset property. If you want to control whether an individual row can be deleted at runtime, you should not set the DeleteEnabled rowset property to False.

For consistency, PeopleSoft recommends that either all rows at a level should disable deletions, or they should all allow deletions.

For rows created with non-Component Processor data (such as message rowsets, Application Engine rowsets, and so on) this property has no effect.

Note. Don't use this property with rows from rowsets created using CreateRowset. Rowsets created using the CreateRowset function are standalone rowsets, not tied to the database, so there are no database updates when they are manipulated. Delete and insert activity on these types of rowsets aren't automatically applied at save time.

This property is read-write.

See Also

PeopleSoft Application Designer, Using Page Controls, Using Scroll Areas and Scroll Bars, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using Standalone Rowsets, Chapter 27, Rowset Class , DeleteEnabled, page 1504.

IsChanged

Description

This property returns True if any field value on the primary database record of the row has been changed.

Note. Use this property only with the primary database record. It doesn't return valid results if used with a work record. This property returns True only if the existing row object has a field that has changed. If a field in a lower level rowset has changed, this property returns False.

This property is read-only.

Considerations Using IsChanged

This property and the IsChanged record property do *not*always return identical values.

If a row in a scroll contains multiple records (such as a primary database record and one or more work records), the IsChanged row property returns True if *any* of the records in the row are changed. The IsChanged record property returns True *only* if a specific record, namely, the primary database record, is changed.

If a row is deleted from a scroll, *only*the primary database record has its IsChanged property marked to False (since the row has been deleted.) Any work records in the row *still*have their IsChanged properties set to True.

Example

```
&tmp =  &ROW.IsChanged;
if &tmp = True then
  Warning("A Field on this row has been changed");
End-If;
```

IsDeleted

Description

This property returns True if the row has been deleted.

This property is read-only.

Note. This property also returns True if the row is new and hasn't been changed.

Example

```
&tmp = &ROW.IsDeleted;
```

IsEditError

Description

This property is True if an error has been found for any field associated with any record in the current row or on any row in any child rowset of the current row after executing the **ExecuteEdits** method. This property can be used with the Field class properties **EditError** (to find out which field is in error), and **MessageSetNumber** and **MessageNumber** to find the error message set number and error message number.

This property is read-only.

Example

The following is an example showing how **IsEditError**, along with the **ExecuteEdits** method could be used:

```
&MSG.ExecuteEdits();
If &MSG.IsEditError Then
&RS = &MSG.GetRowset();
  For &J = 1 to &RS.RowCount
    &ROW = &RS.GetRow(&J);
    If &ROW.IsEditError Then
      &REC = &ROW.GetRecord();
      For &I = 1 to &REC.FieldCount
        If &REC.GetField(&I).EditError Then
          LOG_ERROR();  /* application specific call */
        End-If;
      End-For;
    End-If;
  End-For;
End-If;
```

See Also

Chapter 25, Record Class , ExecuteEdits, page 1448, Chapter 25, Record Class , IsEditError, page 1462, Chapter 12, Field Class , EditError, page 854, Chapter 12, Field Class , MessageNumber, page 863, Chapter 12, Field Class , MessageSetNumber, page 863.

IsNew

Description

This property is True if the row is a new (inserted) row.

This property is read-only.

Example

```
&tmp = &ROW.IsNew;
```

ParentRowset

Description

This property returns a rowset object referencing the rowset containing the row.

This property is read-only.

Example

```
&tmp = &ROW.ParentRowset.RowCount;

/* note that rowcount is a property of the Rowset class */
```

recname

Description

If a record name is used as a property, it accesses the record object with that name. This means the following code:

```
&ROW.somerecname
```

acts the same as

```
&ROW.GetRecord(RECORD.somerecname);
```

This property is read-only.

Example

```
&REC = &ROW.QEPC_LEVEL1_REC;
```

RecordCount

Description

This property returns the number of records in the row. It is defined by the "structure" of the scroll, so it is the same for all rows of the rowset.

This property might be used in conjunction with the **GetRecord** method to write code that examines all records in some way.

This property is read-only.

Example

```
For &I = 1 to &ROW.RecordCount
  &REC = &ROW.GetRecord(&I);
  /* do processing */
End-For;
```

RowNumber

Description

This property returns the row number (starting from 1) of the row within its rowset.

This property is read-only.

Example

The following returns the row number of the current effective-date row.

```
&NUMBER = GetRowset(SCROLL.MYRECORD).GetCurrEffRow().RowNumber;
```

Selected

Description

This property returns True if the row is part of a grid and has been selected either by the user or programmatically.

This property is meaningful in PIA only if the grid or scroll area attribute has been set to provide the selection control. In this case, the property reflects the state of the checkbox or radio button used for selection.

This property is read-write.

Example

The following example determines how many rows in a grid are selected.

```
Function Count_Child_Assets(&GRIDLEVEL1 As Rowset, &NUM_CHILDREN)
   REM *** How many child assets are selected?                  *;
   &GRIDLEVEL1 = GetRowset(SCROLL.PARENT_CHILD_VW);
   For &CHILDROW_COUNT = 1 To &GRIDLEVEL1.ActiveRowCount
      If &GRIDLEVEL1.GetRow(&CHILDROW_COUNT).Selected = True Then
         &NUM_CHILDREN = &NUM_CHILDREN + 1;
      End-If;
   End-For;
End-Function;
```

Style

Description

This property returns the name of the style class if one has been set for the row. You can also use this property to change the style of a row. This property takes a string value.

Note. This property *overrides* only the existing style. It doesn't *change* it. The next time the page is accessed the original style is used.

The PSSTYLE style sheet does *not* contain a default style class for a row. The Style property for all rows is initially NULL (that is, two quotation marks with no space between them ("")). The row assumes the style of the grid alternate row style. Fields in the row assume field styles if set.

Setting the property for a row overrides the grid alternate row style and any page field styles (field styles that were set in PeopleSoft Application Designer). It does *not* override any field styles set (using the Field class Style property) for fields in the row.

Setting the Style property back to NULL (that is, two quotation marks with no space between them ("")) turns off the override.

Note. This property is *not* valid for Windows Client applications.

Example

```
&Scroll = GetLevel0().GetRow(1).GetRowset(Scroll.IC_PC_STYLE_2);

&row_class = &Scroll.GetRow(&row);

&row_class.Style = &style;
```

See Also

Chapter 12, Field Class , Style, page 868, *PeopleSoft Application Designer*, Creating Field Definitions.

Visible

Description

If this property is True, the row is visible if displayed on the current page. This property can be set False to hide the row.

When Visible is set to False to hide a row, the row moves to the end of the rowset. This means that the row number of the row being hidden, and any subsequent rows, *changes* as a result of hiding a row. If the row is later made visible again, it is *not* moved back to its original position, but remains at the end of the rowset. It is just moved to the start of all the other hidden rows.

For example, if there are 4 rows in a rowset and row 2 is hidden (that is, Visible = False), that row now becomes row 4. In order to make that row visible again, row 4 must be set to Visible = True. Alternatively, you can use a row object reference: this remains valid even though the row number of the row may have been changed.

Note. If you use the **Sort** method on a rowset with hidden rows, the hidden rows aren't sorted. Only visible rows are sorted.

You cannot hide rows in the current context of the executing program. This means Visible cannot hide the row containing the executing program, or in a child rowset and be executed against its parent rowset. Place your PeopleCode in a parent rowset and execute it against a child rowset.

Example

```
&ROW.Visible = False;
```

The following code uses the Visible property to determine if a row is visible or not before updating the value of the row.

```
&ROWSET = GetRowset(SCROLL.LD_SHP_INV_VW)

For &I = 1 To &ROWSET.ActiveRowCount
  If &ROWSET(&I).Visible Then
    ITEM_SELECTED.value = "N";
  End-If;
End-For;
```

The following code starts with the last row and works forward to the first. If any changes are made they don’t impact the position of rows that still have to be processed.

```
For &I = &ACTIVE_STREAMS to 1 Step -1
  &Row = &STREAM_ROWSET(&I);
  If &Row.QS_STREAM8.STREAM_ROOT_ID.Value <> &STREAM_ROOT_ID Then
    &Row.Visible = False;
  End-if;
End-For;
```

CHAPTER 27

Rowset Class

This chapter provides an overview of Rowset class and discusses the following topics:

- Shortcut considerations.
- Rowset Object declaration.
- Scope of a Rowset object.
- Rowset class built-in functions.
- Rowset class methods.
- Rowset class properties.

Understanding Rowset Class

A *rowset* object, instantiated from a Rowset class, is a collection of rows associated with buffer data. A component scroll is a rowset. You can also have a level 0 rowset.

If a rowset object is instantiated using **GetRowset** (either the function or one of the methods) the rowset object that is instantiated is populated with data according to the context in which it was instantiated.

If a rowset object is instantiated using the **CreateRowset** function, the rowset object that's instantiated is a standalone rowset. Any records and field references created by this function are initialized to null values, that is, they do *not* contain any data. You can populate this rowset object using the **CopyTo**, **Fill**, or **FillAppend** methods.

Default processing isn't performed on a standalone rowset. In addition, a standalone rowset isn't tied to the Component Processor. When you fill it with data, no PeopleCode runs (for example, RowInsert, FieldDefault, and so on.) Delete and insert activity on these types of rowsets aren't automatically applied at save time. Use standalone rowsets for work records.

You might use a rowset object and the **ActiveRowCount** property to iterate over all the rows of the rowset, or to access a specific row (using the **GetRow** method) or to find the name of the primary record associated with the scroll (**DBName**).

The Rowset class is one of the data buffer access classes.

See Also

PeopleCode Developer's Guide, Using Methods and Built-In Functions, Using Standalone Rowsets

PeopleCode Developer's Guide, Data Buffer Access

Shortcut Considerations

The default method for the Rowset class is GetRow. This means you can specify just a row number, and not use the GetRow method. For example, the following two lines of code are equivalent:

```
&MyRow = GetRowset()(5);

&MyRow = GetRowset().GetRow(5);
```

Data Type of a Rowset Object

Rowset objects are declared as type Rowset. For example,

```
Local Rowset &MYROWSET;
```

Scope of a Rowset Object

A Rowset can be instantiated from PeopleCode or using Java.

This object can be used anywhere you have PeopleCode, that is, in an Application Class, record field PeopleCode, and so on.

You can't pass a rowset object in as part of a Component Interface user-defined method. (Rowsets aren't common data structures outside of a PeopleSoft system.) However, within a user-defined method for a Component Interface you can use rowset objects.

In an Application Engine program, a rowset object is the equivalent of a record object that contains one row and a single record, that is, the State Record. PeopleSoft suggests using the Record object instead of a rowset object to obtain access to the State Record.

Messages have the same structure as rowsets, that is, hierarchical data structures composed of rows, records, and fields.

File objects can have the same structure as rowsets, that is, hierarchical data structures composed of rows, records, and fields.

See Also

PeopleCode Developer's Guide, Using Methods and Built-In Functions, Using Standalone Rowsets

Rowset Class Built-In Functions

Chapter 1, PeopleCode Built-in Functions, CreateRowset , page 107

Chapter 1, PeopleCode Built-in Functions, GetLevel0 , page 253

Chapter 1, PeopleCode Built-in Functions, GetRowset , page 268

Rowset Class Methods

In this section, we discuss the Rowset class methods. The methods are discussed in alphabetical order.

ClearDeletesChanges

Syntax

```
ClearDeletesChanges()
```

Description

Use the **ClearDeletesChanges** method to clear deleted rows and changed from a standalone rowset.

Note. This method works only with standalone rowsets, that is, rowsets created using the CreateRowset function.

This method differs from **Flush** in a number of ways:

- it does not remove *all* rows from the rowset, only deleted rows
- it only applies to standalone rowsets

This method first clears the deleted rows, that is, all rows that have been deleted using DeleteRow are removed from the rowset and their associated buffers are freed.

This method then clears changed rows. That means any changes done on a row (such as field values changed) or newly inserted rows are now propagated into their original state and the changed buffers, if any, are freed.

After executing this method on a standalone rowset, any row that was previously new or changed no longer has that state. The **IsNew** and **IsChanged** properties of a row return false.

This method does *not* do any database updates.

How would you use this method? Suppose you use a standalone rowset to track changes you need to make to some business process or object. After doing the appropriate database updates to reflect changes recorded in the rowset (that is, inserts or deletes or changes), you call this method to clean up the rowset in preparation for further processing. Without this method, newly inserted rows and changed rows preserve their **IsNew** and **IsChanged** status indefinitely, complicating program logic and potentially leading to duplicate inserts or deletes.

Parameters

None.

Returns

None.

Example

```
REM +-------------------------------------------------+;
REM | Function to Update the DB for a Standalone Rowset|;
```

```
REM +-------------------------------------------------+;
Function ProcessDatabaseUpdateforRowset(&rsIn As Rowset)

   For &i = 1 To &rsIn.RowCount

      &rwTMP = &rsIn.GetRow(&i);
      If &rwIn.IsDeleted And
            Not &rwIn.IsNew Then
         &rTMP = &rwIn.GetRecord(1);
         &rTMP.Delete();
      End-If;

      If &rwTMP.IsNew And
            &rwTMP.IsChanged And
            Not &rwTMP.IsDeleted Then

         &rTMP = &rwTMP.GetRecord(1);
         &rTMP.Insert()
      End-If;

      If Not &rwTMP.IsNew And
            &rwTMP.IsChanged And
            Not &rwTMP.IsDeleted Then

         &rTMP = &rwTMP.GetRecord(1);
         &rTMP.Update();
      End-If;

   End-For;

  REM +-----------------------------------------------+;
  REM | Now we need to reset the Rowset flags and     |;
  REM | remove deleted rows              |;
  REM +-----------------------------------------------+;
  &rsIn.ClearDeletesChanges();

End-Function;
```

See Also

Chapter 27, Rowset Class , Flush, page 1489, Chapter 27, Rowset Class , FlushRow, page 1490, Chapter 26, Row Class , IsChanged, page 1473, Chapter 26, Row Class , IsNew, page 1474.

CopyTo

Syntax

CopyTo(*&DestRowset* [, *record_list*])

Where *record_list* is a list of record names in the form:
[RECORD.*source_recname1*, RECORD.*target_recname1*

```
[, RECORD.source_recname2, RECORD.target_recname2]].  .  .
```

Description

The **CopyTo** method copies *from* the rowset executing the method *to* the specified destination rowset, copying *like-named* record fields and subscrolls at corresponding levels.

If pairs of source and destination record names are given, these are used to pair up the records and subscrolls *before* checking for like-named record fields. Then, after copying the named records pairs, this method copies *all* identically named records.

Note. This method does not work for Application Engine state records. If you don't specify *record_list*, *both* the record name *and* the field name have to match exactly for data to be copied from one record field to another. If you specify *record_list*, after the records have been paired up, the field names have to match before any data is copied.

If the rowset you are copying *from* has the field level **EditError**, as well as the **MessageNumber** and **MessageSetNumber** properties set, these values are copied to the rowset you are copying to. For example, suppose you had an application message that had errors. Using the GetSubContractInstance function, these errors would be copied into the message object. From there, you could instantiate a message rowset, copy the message rowset into a work rowset, and use the work rowset to populate Component buffers. (After the field properties are set, the Record, Row, and Rowset properties IsEditError also get set.)

Parameters

Parameter	Description
&DestRowset	Specify the rowset to be copied to. This rowset object must have already been instantiated.
SourceRecname	Specify a record to be copied from, in the rowset object being copied from.
DestRecname	Specify a record to be copied to, in the rowset object to be copied to.

Returns

None.

Example

If you set one rowset equal to another, you haven't made a copy of the rowset. Instead, you have two variables pointing to the *same* data.

To make a clone of an existing rowset, that is, to make two distinct copies, you can do the following:

```
&RS2 = CreateRowset(&RS);
&RS.CopyTo(&RS2);
```

The following example copies data from one rowset object to another. Because no like-named records exist between the two rowsets, the record names are specified. Only the like-named fields are copied from one rowset to the other:

```
Local Rowset &RS1, &RS2;
```

```
Local String &EMPLID;

&RS1 = CreateRowset(RECORD.PERSONAL_DATA);
&RS2 = CreateRowset(RECORD.PER_VENDOR_DATA);
&EMPLID = "8001";
&RS1.Fill("WHERE EMPLID =: 1", &EMPLID);
&RS1.CopyTo(&RS2, RECORD.PERSONAL_DATA, RECORD.PER_VENDOR_DATA);
```

The following example copies data from a message into the Component buffers, then calls the page (using TransferPage) to redraw the page. You could do this to fill a page with message data that is in error, so that an end-user can make corrections to the message data. &WRK_ROWSET0 is the level 0 rowset and &WRK_ROWSET1 is where the data is copied to.

```
/* Get the Message */
&MSG = GetSubContractInstance(&PUBID, &PUBNODE, &CHNLNAME, &MSGNAME, &SUBNAME);

/* Get the Message Rowset */
&MSG_ROWSET = &MSG.GetRowset();

/* Get Level 0 */
&WRK_ROWSET0 = GetLevel0();

/* Create Work rowset */
&WRK_ROWSET1 = GetLevel0()(1).GetRowset(SCROLL.EN_REVISION_TMP);

/* Populate Work Rowset */
&MSG_ROWSET.CopyTo(&WRK_ROWSET1, RECORD.EN_REVISION, RECORD.EN_REVISION_TMP);

SetNextPage("EN_REVISION_MSG");

TransferPage();
```

See Also

Chapter 26, Row Class , CopyTo, page 1467, Chapter 1, PeopleCode Built-in Functions, CreateRowset , page 107, *PeopleCode Developer's Guide*, Understanding Objects and Classes in PeopleCode, Assigning Objects.

DeleteRow

Syntax

DeleteRow(*n*)

Description

The **DeleteRow** method deletes the row in the rowset identified by the parameter.

If the program is being run from a component against Component buffer data, a RowDelete PeopleCode event also fires, followed by the events that normally follow a RowDelete, as if the user had manually pressed ALT+8 and ENTER.

This method initially marks the row as needing to be deleted. At save time the row is actually deleted from the database and cleared from the buffer.

DeleteRow cannot be executed from the same rowset where the deletion takes place, or from a child rowset against a parent. Place your PeopleCode in a parent rowset and execute it against a child rowset.

When **DeleteRow** is used in a loop, you have to process rows from high to low to achieve the correct results, that is, you must delete from the bottom up rather than from the top down. This is necessary because the rows are renumbered after they are deleted (if you delete row one, row two becomes row one).

Note. If you use **DeleteRow** on a rowset created using the CreateRowset function, the row isn't automatically deleted in the database when the page is saved. Rowsets created using the CreateRowset function are standalone rowsets, not tied to the database, so there are no database updates when they are manipulated. Delete and insert activity on these types of rowsets aren't automatically applied at save time.

Parameters

Parameter	Description
n	An integer identifying a row within the rowset object. This must be >= 1 and <= the number of active rows in the rowset (see **ActiveRowCount**).

Returns

An optional Boolean value: True if row is deleted, False otherwise.

Example

In the following example **DeleteRow** is used in a **For** loop. The example checks a value in each row, then conditionally deletes the row. Note the syntax of the **For** loop, including the use of the -1 in the **Step** clause to loop from the highest to lowest values. This ensures that the renumbering of the rows do not affect the loop.

```
For &I = &RS2.ActiveRowCount To 1 Step –1
  If None(&CHECK_SEQ) Then
   &RS2.DeleteRow(&I);
  End-If;
End-For;
```

See Also

Chapter 27, Rowset Class , Flush, page 1489, Chapter 27, Rowset Class , FlushRow, page 1490, Chapter 27, Rowset Class , InsertRow, page 1494, Chapter 25, Record Class , Insert, page 1452, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using Standalone Rowsets.

Fill

Syntax

Fill([*wherestring* [, *bindvalue*] . . .])

Description

The **Fill** method flushes the rowset then reads records from the database into successive rows. The records are read from the database tables corresponding to the primary database record of the scroll into that record. The records are selected by the optional *wherestring* SQL clause, in which the optional *bindvalues* are substituted, using the usual bind placeholders *(:n*).

In general, use this method only with rowsets that were created using the CreateRowset function.

Note. Because Flush always leaves one row in the scroll, there will be one row in the scroll even if you don't read any records.

The actual number of records read into the rowset is an optional return of this method.

Note. This method does not work with Application Engine state records. Also, you cannot use this method in dynamic views.

When this method executes, unlike the **Select** method, it does *not* cause any associated PeopleCode to run as part of reading data into the rowset.

Note. Fill reads only the primary database record. It does not read any related records, nor any subordinate rowset records.

For every record read with the **Fill** method, if the set language is not the base language and the record has related language records, the **Fill** method tries to read the related language record and does related language processing.

The **Fill** method uses a correlation ID of FILL for the table it reads. You must use the correlation ID if you want to refer to the rowset table name as part of the *wherestring.* You receive a runtime error if you use the table name as a column prefix instead of the correlation ID.

Sorting Considerations

Rows come unsorted from the database when using Fill. This is not a problem for SQL server, however, it can be a problem for DB2 and Oracle.

See Chapter 27, Rowset Class , Sort, page 1500.

Parameters

Parameter	Description
wherestring	Specify a SQL WHERE clause to use for selecting records to fill the rowset. This can be a string or a SQL definition.
bindvalue	Specify optional bind variables to be used with the WHERE clause.

Returns

The number of records read into the rowset.

Example

The following example reads all of the QA_MYRECORD records into a rowset, and returns the number of rows read:

```
&RS = CreateRowset(RECORD.QA_MYRECORD);
&NUM_READ = &RS.Fill();
```

The following example reads all of the QA_MYRECORD records that have a MYRECORD field equal to the value of &UVAL into a rowset, and returns the number of rows read:

```
&NUM_READ = &RS.Fill("where MYRECORD = :1", &UVAL);
```

To re-use a WHERE clause for the *wherestring* you can use the SQL repository, and a SQL object.

```
&NUM_READ = &RS.Fill(SQL.MYWHERE, &UVAL);
```

The following gets all the SET_CNTRL_REC rows related to the row on the page, then updates SETID with the value from the page. **Fill** is used with a rowset that was created from a message that was just created, that is, a rowset that was unpopulated.

```
If FieldChanged(SETID) Then
  &MSG = CreateMessage(MESSAGE.SET_CNTRL_REC);
  &MSG_ROWSET = &MSG.GetRowset();
  &MSG_ROWSET.Fill("where SETCNTRLVALUE =:1 and REC_GROUP_ID =:2", SETCNTRLVALUE,⇒
 REC_GROUP_ID);

For &I = 1 To &MSG_ROWSET.ActiveRowCount
  &MSG_ROWSET.GetRow(&I).SET_CNTRL_REC.SETID.Value = SETID;
  &MSG_ROWSET.GetRow(&I).PSCAMA.AUDIT_ACTN.Value = "C";
End-For;

&MSG.Publish();

End-If;
```

The following example uses a correlation ID for the table in the Fill SELECT, to enable the use of correlated subqueries in the WHERE clause, such as the usual effective-date subquery:

```
&RSOWNER_NAME = CreateRowset(RECORD.PERSONAL_D00);
&RSOWNER_NAME.Fill("where SETID=:1 AND EMPLID=:2 AND %EffDtCheck⇒
(PERSONAL_D00,FILL,:3)", &SETID, &EMPLID, &EFFDT);
```

The **Fill** method implicitly uses Fill as an alias for the Rowset record. This is helpful for complex Fill *where* clauses with subqueries.

```
&oprclasscountries = CreateRowset(Record.SCRTY_TBL_GBL);

&oprclasscountries.Fill("WHERE FILL.OPRCLASS = :1 AND NOT EXISTS (SELECT 'X' FROM⇒
 PS_SCRTY_SEC_GBL GBL2 WHERE GBL2.OPRCLASS = FILL.OPRCLASS AND GBL2.COUNTRY =⇒
 FILL.COUNTRY AND GBL2.PNLNAME = :2)", &OPRCLASS, %Component);
```

See Also

Chapter 27, Rowset Class , CopyTo, page 1482, Chapter 27, Rowset Class , Select, page 1495, Chapter 27, Rowset Class , FillAppend, page 1488, Chapter 31, SQL Class , page 1613, Chapter 1, PeopleCode Built-in Functions, CreateRowset , page 107, *PeopleTools Global Technology*, Using Related Language Tables.

FillAppend

Syntax

```
FillAppend([wherestring [, bindvalue] . . .])
```

Description

The **FillAppend** method reads records from the database into successive rows of the rowset starting *after* the last row that exists in the rowset. Unlike **Fill**, it doesn't first flush the rowset.

Note. FillAppend appends rows after the last *active* row in the rowset. If you have deleted rows in the rowset, they will still be the last rows.

When a rowset is selected into, any autoselected child rowsets are also read. The child rowsets are read using a where clause that filters the rows according to the where clause used for the parent rowset, using a subselect.

When a rowset is created using CreateRowset, it contains one empty row. If the rowset hasn't been filled with data, **FillAppend** fills that row also (so you don't have an empty row at the start of your rowset.)

The records are read from the database tables corresponding to the primary database record of the scroll into that record. The records are selected by the optional *wherestring* SQL clause, in which the optional *bindvalues* are substituted, using the usual bind placeholders *(:n*).

In general, use this method only with rowsets that were created using the CreateRowset function.

The actual number of records read into the rowset is an optional return of this method.

Note. This method does not work with Application Engine state records. Also, you cannot use this method in dynamic views.

When this method executes, unlike the **Select** method, it does *not* cause any associated PeopleCode to run as part of reading data into the rowset.

Note. FillAppend eads only the primary database record. It does not read any related records, nor any subordinate rowset records.

For every record read with the **FillAppend** method, if the set language is not the base language and the record has related language records, the **FillAppend** method tries to read the related language record and does related language processing.

The **FillAppend** method uses a correlation ID of FILLAPPEND for the table it reads. You must use the correlation ID if you want to refer to the rowset table name as part of the *wherestring*. You receive a runtime error if you use the table name as a column prefix instead of the correlation ID.

Parameters

Parameter	Description
wherestring	Specify a SQL WHERE clause to use for selecting records to fill the rowset. This can be a string or a SQL definition.
bindvalue	Specify optional bind variables to be used with the WHERE clause.

Returns

The number of records read into the rowset.

Example

The following example reads all of the QA_MYRECORD records that have a MYRECORD field equal to the value of &UVAL into a rowset, and returns the number of rows read:

```
&NUM_READ = &RS.FillAppend("where MYRECORD = :1", &UVAL);
```

To re-use a WHERE clause for the *wherestring* you can use the SQL repository, and a SQL object.

```
&NUM_READ = &RS.FillAppend(SQL.MYWHERE, &UVAL);
```

See Also

Chapter 27, Rowset Class , CopyTo, page 1482, Chapter 27, Rowset Class , Select, page 1495, Chapter 27, Rowset Class , Fill, page 1485, Chapter 31, SQL Class , page 1613, Chapter 1, PeopleCode Built-in Functions, CreateRowset , page 107, *PeopleTools Global Technology*, Using Related Language Tables.

Flush

Syntax

```
Flush()
```

Description

Use the **Flush** method to remove all rows from the rowset and free its associated buffer. Rows that are flushed are not deleted from the database. This function is often used to clear a work scroll before using the Select method.

Note. Flush always leaves one row in the scroll.

You cannot flush the current context of the executing program. This means **Flush** cannot be used for the rowset containing the executing program, or in any child rowset and executed against the parent rowset. Place your PeopleCode in a parent rowset and execute it against a child rowset.

Note. Flush removes all rows and inserts a row. If the rowset is created from component data, the row is initialized (that is, the defaults and RowInit PeopleCode are run, if appropriate.) If the rowset is created from message data, an Application Engine program, and so on, the rows are flushed but the row that is inserted is *not* initialized.

Parameters

None.

Returns

None.

Example

The following example checks for the value of the field CHECKLIST_CD. If something exists in this field, the second level rowset is flushed and new values are selected into it.

```
If All(CHECKLIST_CD) Then
   &RS1H.Flush();
   &RS1H.Select(RECORD.CHECKLIST_ITEM, "where Checklist_CD = :1 and EffDt = ⇒
(Select Max(EffDt) from PS_CHECKLIST_ITEM Where CheckList_CD = :2)", CHECKLIST_CD,⇒
 CHECKLIST_CD);
End-If;
```

See Also

Chapter 27, Rowset Class , FlushRow, page 1490, Chapter 27, Rowset Class , DeleteRow, page 1484, Chapter 27, Rowset Class , InsertRow, page 1494.

FlushRow

Syntax

FlushRow(*n*)

Description

Use the **FlushRow** method to remove a specific row from a rowset and from the Component buffer. Rows that are flushed are not deleted from the database.

FlushRow is a specialized and infrequently used method. In most situations, you want to use **DeleteRow** to remove a row from the Component buffer and remove it from the database as well when the component is saved.

You cannot flush the current context of the executing program. This means **FlushRow** cannot be used for the rowset containing the executing program, or in any child rowset and executed against the parent rowset. Place your PeopleCode in a parent rowset and execute it against a child rowset.

Parameters

Parameter	Description
n	An integer identifying a row within the rowset object. This must be >= 1 and <= the number of rows in the rowset (see property RowCount).

Returns

None.

Example

```
&ROWSET.FlushRow(&ROWNUMBER);
```

See Also

Chapter 27, Rowset Class , DeleteRow, page 1484, Chapter 27, Rowset Class , Flush, page 1489, Chapter 27, Rowset Class , InsertRow, page 1494.

GetCurrEffRow

Syntax

```
GetCurrEffRow()
```

Description

GetCurrEffRow returns a row object for the row with the current effective date. This includes dates equal to the current date, but not dates in the future. If the primary record associated with the rowset is not effective-dated this method returns a null object.

Newly inserted rows are intentionally skipped when looking for the current effective-dated row. This is to find the current effective row from the data that already exists in the database. In other words, a row cannot be considered as a current effective row until it is saved. To find the current effective-dated row in data that has been newly inserted and not yet saved, use the GetNewEffRow method instead.

Parameters

None.

Returns

A row object for the row with the current effective date.

Example

```
&tmp =  &ROWSET.GetCurrEffRow().RowNumber;
/* note RowNumber is a property of the row class */
```

See Also

Chapter 27, Rowset Class , DeleteEnabled, page 1504, Chapter 27, Rowset Class , GetRow, page 1492, Chapter 26, Row Class , GetNextEffRow, page 1467, Chapter 27, Rowset Class , GetNewEffRow, page 1491, Chapter 1, PeopleCode Built-in Functions, GetRow , page 267.

GetNewEffRow

Syntax

```
GetNewEffRow()
```

Description

GetNewEffRow returns a row object for the row with the effective date *closest* to the current date, including newly inserted rows. This means dates equal to the current date, but not future dates. If the primary record associated with the rowset is not effective-dated this method returns a null object.

To find the current effective-dated row from the data that already exists in the database, that is, only from saved rows and *not* a newly created row, use the **GetCurEffRow** method instead.

Parameters

None.

Returns

A row object for the row with the current effective date.

Example

```
Local Row &MyRow = &ROWSET.GetNewEffRow();
```

See Also

Chapter 27, Rowset Class , GetRow, page 1492, Chapter 26, Row Class , GetNextEffRow, page 1467, Chapter 27, Rowset Class , GetCurrEffRow, page 1491, Chapter 1, PeopleCode Built-in Functions, GetRow , page 267, Chapter 27, Rowset Class , DeleteEnabled, page 1504.

GetRow

Syntax

GetRow(*n*)

Description

GetRow returns a row object from a rowset. This is a *default method* for rowsets. This means that any rowset object, followed by a parameter list, acts as if GetRow is specified.

Parameters

Parameter	Description
n	An integer identifying a row within the rowset object. This must be >= 1 and <= the number of rows in the rowset (see property RowCount).

Returns

Returns a row object for the specified row of the rowset.

Example

In the following example, all of the lines of code do the same thing: they return the 5th row from the rowset (&ROWSET is a rowset object.)

```
&ROW = GetRowset().GetRow(5);
&ROW = GetRowset()(5);
&ROW = &ROWSET.GetRow(5);
&ROW = &ROWSET(5);
```

The following example loops through all the rows in a rowset:

```
Local rowset &LEVEL1;
Local row &ROW;
```

```
&LEVEL1 = GetRowset(SCROLL.EMPL_CHECKLIST);
For &I = 1 to &LEVEL1.ActiveRowCount
  &ROW = &LEVEL.GetRow(&I);
  /* do some processing */
End-For;
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetRow , page 267.

HideAllRows

Syntax

HideAllRows()

Description

HideAllRows hides all rows of the rowset. Using this method is equivalent to a loop setting the visible property of each row of the rowset to False.

You cannot hide rows in the current context of the executing program. This means **HideAllRows** cannot hide the rowset containing the executing program, or in any child rowsets and be executed against the parent rowset. Place your PeopleCode in a parent rowset and execute it against a child rowset.

Parameters

None.

Returns

None.

Example

The following example hides the second level scroll if a value exists in the NO_COMMENTS field in the first level of the scroll. The code is running from the first level of the scroll.

```
Local Rowset &RS1, &RS2;

&RS1 = GetRowset();
&RS2 = GetRowset(SCROLL.EMPL_CHKLST_ITM);

For &I = 1 to &RS1.ActiveRowCount
  If ALL(&RS1.GetRow(&I).EMPLOYEE_CHECKLIST.NO_COMMENTS) Then
   &RS2.HideAllRows();
  End-If;
/*other processing */
End-For;
```

See Also

Chapter 27, Rowset Class , ShowAllRows, page 1500, Chapter 26, Row Class , Visible, page 1477.

InsertRow

Syntax

InsertRow(*n*)

Description

InsertRow programmatically performs the ALT+7 and ENTER (**RowInsert**) function. InsertRow inserts a new row in the current rowset *after* the row specified by the parameter if the primary record for the rowset is not effective-dated or a standalone rowset. If the primary record for the rowset is effective-dated or a standalone rowset, the new row is inserted *before* the current row, and all values from the current row are copied into the new row, except for EFFDT, which is set to the current date.

In addition, **InsertRow** propagates the keys from the higher level (if any) into the inserted row.

If the program is being run from a component against Component buffer data, a RowInit PeopleCode event also fires, followed by the events that normally follow a RowInsert, as if the user had manually pressed ALT+7 and ENTER.

For rowsets corresponding to Component buffer data, the **InsertRow** method only executes in turbo mode: that is, default processing is performed, but only on the row being inserted.

InsertRow cannot be executed from the same rowset where the insertion will take place, or from a child rowset against a parent. Place your PeopleCode in a parent rowset and execute it against a child rowset.

Note. If you use **InsertRow** on a rowset created using the CreateRowset function, the row isn't automatically inserted in the database when the page is saved. Rowsets created using the CreateRowset function are standalone rowsets, not tied to the database, so there are no database updates when they are manipulated. Delete and insert activity on these types of rowsets aren't automatically applied at save time.

Effective-Dated Row Considerations

When a row is inserted, if that row contains child scrolls, this method also inserts an empty row for any child scrolls. The effective-date field for this empty row is also empty. The current date is *not* used.

Parameters

Parameter	Description
n	An integer identifying a row within the rowset object. This must be >= 0 and <= the number of active rows in the rowset (see property ActiveRowCount). A value of 0 inserts in front of the first row.

Returns

An optional Boolean value: True if the row is inserted, False if the row is not inserted.

Example

In the following example, as the primary database record isn't effective-dated, the new row is inserted *after* the second row in the rowset.

```
&ROWSET.InsertRow(2);
```

See Also

Chapter 27, Rowset Class , DeleteRow, page 1484, Chapter 25, Record Class , Delete, page 1447, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using Standalone Rowsets.

Refresh

Syntax

```
Refresh()
```

Description

Refresh reloads the rowset object from the database using the current page keys. This causes the page to be redrawn. GetLevel0().Refresh() refreshes the entire page. If you only want a specific scroll to be redrawn, you can use the refresh method with that rowset. You don't have to use a level 0 rowset with this method. This is particularly useful after using RemoteCall.

Note. If a scroll is marked as No Auto Select in Application Designer, **Refresh** will not work on it. The **Refresh** method clears the existing buffers and does a refresh from the database. You do *not* want to use this method if your rowset is based on message data. Instead, you can use the CopyTo Rowset method combined with TransferPage.

Parameters

None.

Returns

None.

Example

The following example refreshes the entire page.

```
&Mylevel0 = GetLevel0().Refresh();
```

The following example refreshes only the second level scroll of the page:

```
&RowSetLevel2.Refresh();
```

See Also

Chapter 27, Rowset Class , CopyTo, page 1482, Chapter 1, PeopleCode Built-in Functions, RemoteCall , page 364, Chapter 1, PeopleCode Built-in Functions, TransferPage , page 475.

Select

Syntax

```
Select([parmlist], RECORD.selrecord [, wherestr, bindvars])
```

Where *paramlist* is a list of child rowsets, given in the following form:

```
SCROLL.scrollname1 [, SCROLL.scrollname2] . . .
```

The first *scrollname* must be a child rowset of the rowset object executing the method, the second *scrollname* must be a child of the first child, and so on.

Description

Select, like the related method **SelectNew**, reads data from the database tables or views into either a row or rowset object.

Note. This method is valid only when used with rowsets that reference data from the Component buffers. You cannot use this method with a message rowset, a rowset from an Application Engine state record, and so on. You can't use this method with rowsets created using the CreateRowset function (use the **Fill** method instead.)

Select automatically places child rowsets in the rowset object executing the method under the correct parent row. If it cannot match a child rowset to a parent row, an error occurs.

When a rowset is selected into, any autoselected child rowsets is also read. The child rowsets are read using a where clause that filters the rows according to the where clause used for the parent rowset, using a subselect.

If you don't specify any child rowsets in *paramlist*, **Select** selects from a SQL table or view specified by *selrecord* into the rowset object executing the method.

If you specify a child rowset in *paramlist*, **Select** selects from a SQL table or view specified by *selrecord* into the child rowset specified in *paramlist*, under the appropriate row of the rowset executing the method.

If the rowset executing the method is a level 0 rowset, and you specify **Select** without specifying any child rowsets with *paramlist*, the method reads only a single row, because only one row is allowed at level 0.

The rowset executing the method *does not* have to be a level 0 rowset, so the **Select** method can produce the functionality of both the ScrollSelect and RowScrollSelect functions.

Note. *For developers familiar with previous releases of PeopleCode.* If the rowset executing the method is a level 0 rowset, and you specify a child rowset with *paramlist*, this method functions exactly like **ScrollSelect**. If the rowset executing the method is not a level 0 rowset, and no child rowsets are specified with *paramlist*, the method acts like **RowScrollSelect**.

The record definition of the table or view being selected from is called the *select record*, and identified with RECORD.*selrecord*. The select record can be the same as the primary database record associated with the rowset, or it can be a different record definition that has compatible fields. If you define a select record that differs from the primary database record for the rowset, you can restrict the number of fields that are loaded into the buffers on the client workstation by including only the fields that you actually need.

Select accepts a SQL string that can contain a WHERE clause restricting the number of rows selected into the object. The SQL string can also contain an ORDER BY clause to sort the rows.

Select and its related methods generate a SQL SELECT statement at runtime, based on the fields in the select record and the WHERE clause passed to them in the method parameters.

Parameters

Parameter	Description
paramlist	An optional parameter list, for specifying children rowsets of the rowset executing the method, as the rowset to be selected into. The first *scrollname* in *paramlist* must be a child rowset of the rowset executing the method, the second *scrollname* must be a child of the first child, and so on.
RECORD.*selrecord*	Specifies the select record. The *selrecord* record must be defined in PeopleSoft Application Designer and be a build SQL table (using **Build, Project**) as a table or a view, unless *selrecord* is the same record as the primary database record associated with the rowset. *selrecord* can contain fewer fields than the primary record associated with the rowset, although it must contain any key fields to maintain dependencies with other records.
wherestr	Contains a WHERE clause to restrict the rows selected from *selrecord* and/or an ORDER BY clause to sort the rows. The WHERE clause can contain the PeopleSoft meta-SQL functions such as %DateIn() or %CurrentDateIn. It can also contain inline bind variables.
bindvars	A list of bind variables to be substituted in the WHERE clause. The same restrictions that exist for SQLExec exist for these variables. See SQLExec for further information.

Returns

The number of rows read (optional.) This counts only the lines read into the specified rowset. It does not include any additional rows read into any child rowsets of the rowset.

Example

The following example selects into the level 0 rowset, only one row. Depending on the autoselect settings on any child scrolls, they may be reselected too.

```
&LEVEL0 = GetLevel0();
&LEVEL0.Select(RECORD.PERSONAL_DATA, "WHERE.  .  .");
```

The following example selects into the level 1 scroll specified. Depending on the autoselect settings on any child (level 2) scrolls, they may also be selected.

```
&LEVEL0.Select(SCROLL.EMPL_CHECKLIST, RECORD.EMPL_CHECKLIST, "WHERE.  .  .");
```

The following example selects into the child scroll of the level 1 rowset. Each row fetched is placed under the appropriate row in &LEVEL1. Note that instead of hard-coding the WHERE clause, the SQL repository is used to access a SQL definition named SELECT_WHERE.

```
&LEVEL1 = &LEVEL0()(1).GetRowset(SCROLL.EMPL_CHECKLIST);
&LEVEL1.Select(SCROLL.EMPL_CHKLST_ITM, RECORD.EMPL_CHKLST_ITM, SQL.SELECT_WHERE);
```

The following example first flushes the hidden work scroll, then selects into it based on a field on the page.

```
&RS1H.Flush();
&RS1H.Select(RECORD.CHECKLIST_ITEM, "where Checklist_CD = :1 and EffDt = (Select⇒
 Max(EffDt) from PS_CHECKLIST_ITEM Where CheckList_CD = :2)", CHECKLIST_CD,⇒
 CHECKLIST_CD);
```

See Also

Chapter 27, Rowset Class , SelectNew, page 1498, Chapter 25, Record Class , SelectByKey, page 1454.

SelectNew

Syntax

```
SelectNew([parmlist], RECORD.selrecord [, wherestr, bindvars])
```

Where *paramlist* is a list of child rowsets, given in the following form:

```
SCROLL.scrollname1 [, SCROLL.scrollname2] . . .
```

The first *scrollname* must be a child rowset of the rowset executing the method, the second *scrollname* must be a child of the first child, and so on.

Description

SelectNew is similar to **Select**, except that all rows read into the rowset are marked *new* so they are automatically inserted into the database at Save time. This capability can be used, for example, to insert new rows into the database by selecting data using a view of columns from other database tables.

Note. This method is valid only when used with rowsets that reference data from the Component buffers. You cannot use this method with a message rowset, a rowset from an Application Engine state record, and so on. You can't use this method with rowsets created using the CreateRowset function (use the **Fill** method instead.)

Parameters

Parameter	Description
paramlist	An optional parameter list, for specifying children rowsets of the rowset executing the method, as the rowset to be selected into. The first *scrollname* in *paramlist* must be a child rowset of the rowset executing the method, the second *scrollname* must be a child of the first child, and so on.
RECORD.*selrecord*	Specifies the select record. The *selrecord* record must be defined in PeopleSoft Application Designer and be a build SQL table (using **Build, Project**) as a table or a view, unless *selrecord* is the same record as the primary database record associated with the rowset. *selrecord* can contain fewer fields than the primary record associated with the rowset, although it must contain any key fields to maintain dependencies with other records.
wherestr	Contains a WHERE clause to restrict the rows selected from *selrecord* and/or an ORDER BY clause to sort the rows. The WHERE clause can contain the PeopleSoft meta-SQL functions such as %DateIn() or %CurrentDateIn. It can also contain inline bind variables.
bindvars	A list of bind variables to be substituted in the WHERE clause. The same restrictions that exist for SQLExec exist for these variables. See SQLExec for further information.

Returns

The number of rows read (optional.) This counts only the lines read into the specified rowset. It does not include any additional rows read into any child rowsets of the rowset.

Example

The following example selects rows from DATA2 and reads them into a level one scroll. If the user saves the page, these rows are inserted into DATA1.

```
&LEVEL0.SelectNew(SCROLL.DATA1, RECORD.DATA2, "Where SETID = :1 and CUST_ID =:2",⇒
 CUSTOMER.SETID, CUSTOMER.CUST_ID);
```

If you use the same WHERE clause in more than one Select, you may want to use the SQL repository to store the SQL fragment. That way, if you ever want to change it, you will have to change it in only one place.

```
&LEVEL0.SelectNew(SCROLL.DATA1, RECORD.DATA2, SQL.Select_New);
```

See Also

Chapter 27, Rowset Class , Select, page 1495, Chapter 25, Record Class , SelectByKey, page 1454.

SetDefault

Syntax

SetDefault(*recname.fieldname*)

Description

The **SetDefault** method sets the value of the specified *recname.fieldname* to a null value in every row of the rowset. If this method is being run from a component against Component buffer data, the next time default processing is performed, this value is reinitialized to its default value: either a default specified in its record field definition or one set programmatically by PeopleCode located in a FieldDefault event. If neither of these defaults exist, the Component Processor leaves the field blank.

No default processing is performed against data that is not in the Component buffers.

Blank numbers correspond to zero on the database. Blank characters correspond to a space on the database. Blank dates and long characters correspond to NULL on the database. **SetDefault** gives each field data type its proper value.

Parameters

Parameter	Description
recname.fieldname	Specifies a field. The field must be in the specified record, on the rows of the rowset.

Returns

None.

Example

This example resets the PROVIDER to its null value. This field is reset to its default value when default processing is next performed:

```
If COVERAGE_ELECT = "W" Then
  &ROWSET.SetDefault(INS_NAME_TBL.PROVIDER);
End-if;
```

See Also

Chapter 12, Field Class , SetDefault, page 851, *PeopleCode Developer's Guide*, PeopleCode and the Component Processor, Default Processing.

ShowAllRows

Syntax

```
ShowAllRows()
```

Description

ShowAllRows "unhides" all rows of the rowset object executing the method. Using this method is equivalent to a loop setting the visible property of each row of the rowset to True.

ShowAllRows cannot be executed from the same rowset you want to display, or from a child rowset against a parent. Place your PeopleCode in a parent rowset and execute it against a child rowset.

Parameters

None

Returns

None.

Example

```
&ROWSET.ShowAllRows();
```

See Also

Chapter 27, Rowset Class , HideAllRows, page 1493, Chapter 26, Row Class , Visible, page 1477.

Sort

Syntax

```
Sort([paramlist,] sort_fields)
```

Where *paramlist* is a list of child rowsets, given in the following form:

```
SCROLL.scrollname1 [, SCROLL.scrollname2] . . .
```

The first *scrollname* must be a child rowset of the rowset executing the method, the second *scrollname* must be a child of the first child, and so on.

And where *sort_fields* is a list of field specifiers in the form:

```
[recordname.]field_1, order_1 [, [recordname.]field_2, order_2]_
```

Description

Sort programmatically sorts the rows in a rowset. The rows can be sorted on one or more fields.

If you specify a child rowset in *paramlist*, **Sort** selects a set of child rowsets to be sorted. If you don't specify a child rowset in *paramlist*, the rowset executing the method is sorted.

The type of sort done by this function, that is, whether it is a linguistic or binary sort, is determined by the Sort Order Option on the PeopleTools Options page.

Note. The Sort method sorts only *visible* rows in a rowset. If the rowset you're sorting has hidden rows, the hidden rows are *not* sorted.

Parameters

Parameter	Description
paramlist	An optional parameter list, for specifying children rowsets of the rowset executing the method, as the rowset to be sorted. The first *scrollname* in *paramlist* must be a child rowset of the rowset executing the method, the second *scrollname* must be a child of the first child, and so on.
sort_fields	A list of field and order specifiers which act as sort keys. The rows in the rowset will be sorted by the first field in either ascending or descending order, then by the second field in either ascending or descending order, and so on.
[*recordname*.]*field_n*	Specifies a sort key field in the rowset. The *recordname* prefix is required if you've specified a field that is not on the current record.
order_n	A single-character string. "A" specifies ascending order; "D" specifies descending order.

Returns

None.

Example

The first example repopulates a rowset in a page programmatically by first flushing its contents, selecting new contents using **Select**, then sorting the rows in ascending order by EXPORT_OBJECT_NAME:

```
Function populate_rowset;

  &RS1 = GetLevel0()(1).GetRowset(SCROLL.EXPORT_OBJECT);
  &RS1.Flush();
  &RS1.Select(RECORD.EXPORT_OBJECT, "where export_type =⇒
:EXPORT_TYPE_VW.EXPORT_TYPE");
  &RS1.Sort(EXPORT_OBJECT_NAME, "A");

End-Function;
```

See Also

Chapter 27, Rowset Class , Select, page 1495, Chapter 26, Row Class , GetRowset, page 1470, Chapter 26, Row Class , GetRowset, page 1470.

Rowset Class Properties

In this section, we discuss the Rowset class properties. The properties are discussed in alphabetical order.

ActiveRowCount

Description

This property returns the number of active (non-deleted) rows in the rowset.

This property returns a value of 1 for an empty scroll (**Flush** always leaves an empty row.) You can use the **IsChanged** or **IsNew** properties to check the row.

This property is read-only.

Example

```
&tmp = &ROWSET.ActiveRowCount;
```

ChangeOnInit

Description

Normally, if a field value is changed, whether through PeopleCode or by an end-user, the IsChanged property for the row is set to True. The exception to this is when a change is done in the FieldDefault or FieldFormula event, that is, if a value is set in one of these events, the row is not marked as changed.

At save time, all newly inserted *and* changed rows are written to the database. All newly inserted but *not* changed rows are *not* written to the database.

Use this property to specify whether a change made to a new row from RowInit or RowInsert PeopleCode is marked as changed. This property takes a Boolean value. The default value is True.

Setting this property to False causes changes to fields for the rowset done in RowInit and RowInsert PeopleCode to *not* mark a new row as IsChanged.

This property is read-write.

Runtime Considerations

You must set this property *before*anything is changed for a row, otherwise the row is marked as changed.

This property propagates to child rowsets. That is, if it's set for a parent rowset before doing an insert row, it's automatically set on any and all child rowsets.

In addition, parent rows are marked as changed if this property is set only for the child rowset. That is, if you've changed a level 3 child of a level 2 row, the level 2 and level 1 rows are marked as changed to maintain data integrity.

DataAreaCollapsed

Description

This property specifies whether the view of a data area is collapsed or expanded.

Note. You must set the Collapsible Data Area field on the properties for the level-based control for this property to have any effect.

This property changes to reflect the current state of the data area, according to whether the user has collapsed or expanded it. Changing the value collapses or expands the data area, but it does *not* prevent the user from collapsing (or expanding) it themselves.

Note. Because the user can change the value of this property, whatever value is set in PeopleCode isn't guaranteed to be still set the next time it is checked, because the user may have collapsed or expanded the data area in the meantime.

This property overwrites the value of the Default Initial View to Expanded field set in PeopleSoft Application Designer. For example, if Default Initial View to Expanded is selected in PeopleSoft Application Designer, then the value for the DataAreaCollapsed property is set to True, the control initially displays collapsed.

This property takes a Boolean value: True, initially display the data area collapsed, False, initially display the data area expanded.

This property is read-write.

Note. To collapse just a group box, use the DataAreaCollapsed field method.

See Also

Chapter 12, Field Class , DataAreaCollapsed, page 852.

Example

The following example checks the number of rows in the level 1 rowset. If the number of rows returned is larger than 100, the data area is initially displayed collapsed.

```
If &Level1.RowCount > 100 Then
  &Level1.DataAreaCollapsed = True;
Else
  &Level1.DataAreaCollapsed = False;
End-If;
```

DBRecordName

Description

This property returns the name of the primary database record as a string for the rowset.

This property is read-only.

Example

```
&DBNAME = &ROWSET.DBRecordName;
```

DeleteEnabled

Description

This property determines whether a rowset allows rows to be deleted (the equivalent of the user pressing ALT+8 and ENTER). This property takes a Boolean value.

Note. This property controls only whether an end-user can delete a row. Rows can still be deleted using PeopleCode.

The initial value of this property depends on how the scroll was created at design time. If the No Row Delete setting is selected on the Use tab of the scroll properties dialog box, DeleteEnabled will be False: otherwise it will be True.

Note. If No Row Delete is selected in PeopleSoft Application Designer, setting DeleteEnabled to True will *not* override this value. To control whether a rowset allows deletions at runtime, you should *not* select No Row Delete at design time.

For consistency, PeopleSoft recommends that either all rowsets at a level should disable deletions, or they should all allow deletions.

For rowsets created with non-Component Processor data (such as message rowsets, Application Engine rowsets, and so on) this property has no effect.

Note. Don't use this property with rowsets that are created using CreateRowset. Rowsets created using the CreateRowset function are standalone rowsets, not tied to the database, so there are no database updates when they are manipulated. Delete and insert activity on these types of rowsets aren't automatically applied at save time.

This property is read-write.

See Also

PeopleSoft Application Designer, Using Page Controls, Using Scroll Areas and Scroll Bars, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using Standalone Rowsets, Chapter 27, Rowset Class , DeleteEnabled, page 1504, Chapter 27, Rowset Class , InsertEnabled, page 1505.

EffDt

Description

This property references the effective date of the primary record associated with the rowset. If the primary record associated with the rowset is not effective-dated, this property has a null value. To find the primary record associated with a rowset object, you can use the **DBRecordName** property.

Note. This property isn't valid with rowsets created using the CreateRowset function.

This property is read-only.

Example

```
&tmp = &ROWSET.EffDt;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateRowset , page 107.

EffSeq

Description

This property references the effective-sequence number of the primary record associated with the rowset. If the primary record associated with the rowset does not have an effective-sequence number, this property has the value 0. To find the primary record associated with a rowset object, you can use the **DBRecordName** property.

Note. This property isn't valid with rowsets created using the CreateRowset function.

This property is read-only.

Example

```
&tmp = &ROWSET.EffSeq;
```

See Also

Chapter 1, PeopleCode Built-in Functions, CreateRowset , page 107.

InsertEnabled

Description

This property determines whether a rowset allows rows to be inserted (the equivalent of the user pressing ALT+7 and ENTER). This property takes a Boolean value.

Note. This property controls only whether an end-user can insert a row. Rows can still be inserted using PeopleCode.

The initial value of this property depends on a value set at design time. If the No Row Insert setting is selected on the Use tab of the scroll properties dialog box, InsertEnabled is False: otherwise it is True.

Note. If No Row Insert is selected in PeopleSoft Application Designer, setting InsertEnabled to True does *not* override this value. To control whether a rowset allows inserts at runtime, you should *not* select No Row Insert at design time.

For consistency, PeopleSoft recommends that either all rowsets at a level should disable inserts, or they should all allow inserts.

For rowsets created with non-Component Processor data (such as message rowsets, Application Engine rowsets, and so on) this property has no effect.

Note. Don't use this property with rowsets created using CreateRowset. Rowsets created using the CreateRowset function are standalone rowsets, not tied to the database, so there are no database updates when they are manipulated. Delete and insert activity on these types of rowsets aren't automatically applied at save time.

This property is read-write.

See Also

Chapter 27, Rowset Class , DeleteEnabled, page 1504, *PeopleSoft Application Designer*, Using Page Controls, Using Scroll Areas and Scroll Bars, *PeopleCode Developer's Guide*, Using Methods and Built-In Functions, Using Standalone Rowsets.

IsEditError

Description

This property is True if an error has been found on any field in any record in any row or child rowset of the current rowset after executing the **ExecuteEdits** method on either a message object or a record object. This property can be used with the Field Class properties **EditError** (to find the field that's in error), **MessageSetNumber** and **MessageNumber** to find the error message set number and error message number.

This property is read-only.

Example

The following is an example showing how **IsEditError**, along with **ExecuteEdits** could be used:

```
&REC.ExecuteEdits();
If &ROWSET.IsEditError Then
  For &I = 1 to &ROWSET.ActiveRowCount
    &ROW = &ROWSET(&I);
    For &J to &ROW.RecordCount
      &REC = &ROW.GetRecord(&J);
      For &K = 1 to &REC.FieldCount
        If &REC.GetField(&K).EditError Then
          LOG_ERROR();
          /* application specific call */
        End-If;
      End-For;
    End-For;
  End-For;
End-If;
```

See Also

Chapter 25, Record Class , ExecuteEdits, page 1448, Chapter 25, Record Class , IsEditError, page 1462, Chapter 12, Field Class , EditError, page 854, Chapter 12, Field Class , MessageNumber, page 863, Chapter 12, Field Class , MessageSetNumber, page 863.

Level

Description

This property returns the level, that is, the nesting depth, of the rowset object. The top-level rowset has a level number of 0.

This property is read-only.

Example

```
&tmp = &ROWSET.Level;
```

Name

Description

This property refers to the name of the rowset. This property returns different values, based on the type of rowset.

Type of Rowset	Returns
Rowset created using GetLevel0	returns an empty string
Component buffer rowset	returns primary record name
Message rowset	returns primary record name
Component Interface rowset	returns primary record name
Application Engine rowset	always returns AESTATE

This property is read-only.

Example

```
&tmp = &ROWSET.Name;
```

ParentRow

Description

This property returns a row object containing a reference to the parent row, that is, the row containing the rowset. If this is a top-level rowset (level 0), the **ParentRow** property has a null value.

This property is read-only.

Example

```
&tmp = &ROWSET.ParentRow.RowNumber;
/* note that RowNumber is a property of the row class */
```

ParentRowset

Description

This property returns a rowset object containing a reference to the parent rowset, that is, the rowset containing the rowset. If this is a top-level rowset (level 0), the **ParentRowset** property has a null value.

This property is read-only.

Example

```
&tmp = &ROWSET.ParentRowset.Level;
/* note Level is another property of the rowset class */
```

RowCount

Description

This property returns the total number of rows in the rowset. It includes deleted rows. (The **ActiveRowCount** property doesn't include deleted rows.)

This property is read-only.

Example

```
&tmp = &ROWSET.RowCount;
```

TopRowNumber

Description

This property returns the row that is being displayed at the top of the scroll (if any) for the rowset.

Generally, this property is used to return the top row number of a scroll. However, sometimes you want to reposition the scroll. For example, if you use SetCursorPos to move the focus to a given field, there is no guarantee that the row containing the field is at the top of the scroll.

This property is read-write.

CHAPTER 28

Search Classes

This chapter provides an overview of Search classes and discusses the following topics:

- Search class hierarchy
- Error handling
- Search object declaration
- Scope of the search objects
- Session class method
- Search class reference section

See Also

PeopleSoft Server Tools Administration, Building and Maintaining Search Indexes

Understanding Search Class

The Search API provides enables you to create a logical search index as well as access a search index and query its contents. Every search index in the PeopleSoft system must have a logical search index created to describe it.

The SearchIndex objects can be created for the following types of content:

- Indexing content references in a PeopleSoft portal
- Indexing data in PeopleSoft records
- Spidering a file system
- Spidering URL's
- User defined indexes (such as a PeopleSoft application)

Portal users can index and search on content references through the Search API using PeopleCode. The Portal Administration pages provide a GUI access to indexing a portal. This search index is what is used when doing searches on the portal. Users can create search indexes and index data in PeopleSoft records. The Search Index Designer provides a GUI tool to create these indexes.

Users can create search indexes and spider files systems or URL's. The Search Index Designer provides a GUI tool to create these indexes.

Users can create their own custom search indexes.

PeopleSoft has integrated the Verity search engine into PeopleTools.

All searches done using the Search API are executed against a Verity search index. A search index is similar to a record in many ways:

- It has fields that hold information about a document.
- It stores one row of information for each document indexed.
- It can store information in different languages.

You must build a search index before you can use the Search API.

For the portal, you can build a search index either using the Search API (BuildSearchIndex) PortalRegistry method) or using the Portal Administration.

To index and search something that isn't registered in the portal, you must build your own search index.

After you've built a search index, you can execute a search against the index, and execute a search query. A search query (SearchQuery object) represents the items in an index that match a query. These matched items are retrieved in the SearchResult collection.

In addition, you can also check for any errors that were generated by the search.

See Also

Chapter 28, Search Classes , BuildSearchIndex, page 1516

Using the SearchResults Collection

You specify the size of the SearchResult collection when you execute the search. If more SearchResults were returned than can be contained in the single SearchResult collection, you can access the next set of SearchResults by executing the search again, specifying only a different starting point.

For example, suppose your search returns 28 documents that match the query, but the size you specified when you executed the search was 10. You have to execute the search three times to access all the documents that matched the query.

The following pseudo-code could be behind a "Next Matches" button.

- &Start indicates at what document you want to start your search query.
- &Size is the number of search results returned in every search.
- &SearchQuery is the query object you created at the start of your program.

```
&Start = &Start + &Size;

&SearchQuery.Execute(&Start, &Size);
```

Understanding SearchResults

Each SearchResult in the SearchResult collection contains:

- Key
- Score
- SearchFields

The key uniquely identifies each document in the SearchResult collections. Keys provide a link back to the item that was indexed. The value returned by the Key property depends on the search index. If you're using a search index with the portal, the key contains the content provider and the URL. If you're not using the portal search index, the key is determined by the developer who created the search index.

Each query returns matched documents in relevance-ranked order, with those documents considered most relevant appearing at the top of the list. During search processing, a score is calculated for each retrieved document. This is the value stored in the Score property. A Score is assigned a value from 0.00 to 1.00, where 1.00 represents a perfect match to the search criteria provided.

Each SearchResult contains fields. If this is a non-portal search, these fields were specified by the developer when they created the search index. If this is a portal search, the fields are the following:

Field	Description
VALID_FROM_DATE	Valid from date
VALID_TO_DATE	Valid to date
CREATION_DATE	Creation date
CONTENT_PROVIDER	Content provider name
URL	URL
DESCRIPTION	Description

An external URL has the full URI attached so it might look like this:

```
URL: http://sports.mysports.com/nba/teams/chi/
```

Whereas a URL with a content provider specified is comprised of two portions.

```
Content Provider: HRMS

URL:ICType=Panel&Menu=ADMINISTER_WORKFORCE_(U.S.)&Market=GBL&PanelGroupName=⇒
ABSENCE_HISTORY1
```

You can interrogate each field in the SearchField collection to find its name and value.

See Also

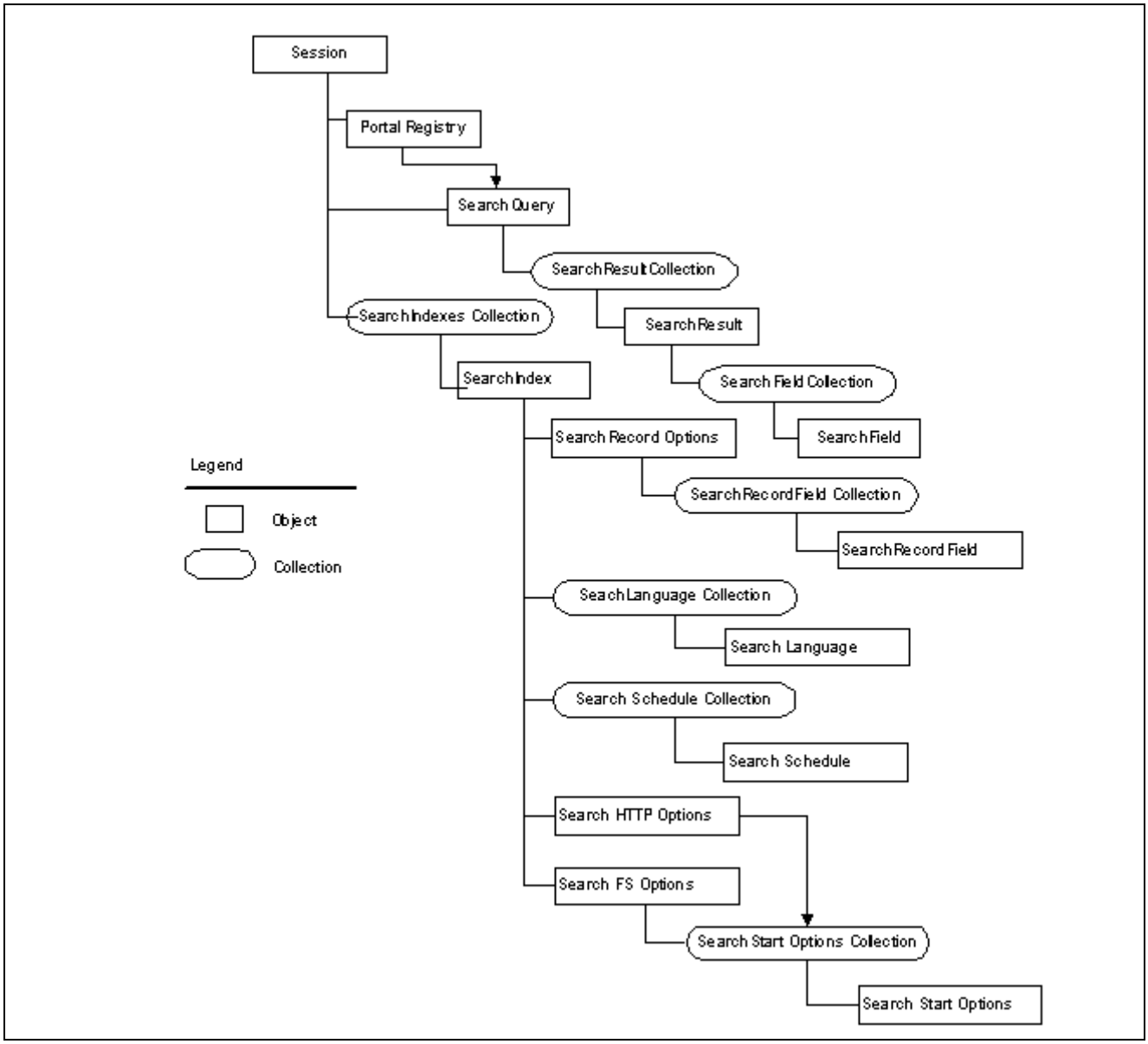
Chapter 28, Search Classes , Key, page 1524

Chapter 28, Search Classes , Score, page 1525

Chapter 28, Search Classes , SearchFields, page 1525

Search Classes Hierarchy

There are many different classes used with the Search API. The following flowchart shows the different classes, and how they are interrelated.



Search API classes hierarchy

Error Handling

All errors for the Search API, like the other APIs, are logged in the PSMessages collection, instantiated from a session object.

The search classes log errors "interactively", that is, as they happen. For example, suppose you specified an invalid SearchField name. The error would be logged in the PSMessages collection as soon as you executed the ItemByName method.

If you want to search for errors in the query string before you execute the search, you can use the SearchQuery **Parse** method.

When to check for errors depends on your application. However, if you check for errors after every assignment, you may see a performance degradation.

The easiest way to check for errors is to check the number of messages in the PSMessages collection, using the *Count* property. If the Count is 0, there are no errors.

```
Local ApiObject &MySession;
Local ApiObject &ERRORCOL;
Component ApiObject &SearchQuery, &SearchResultCol, &SearchResult;
Component Number &Start, &Size;

&MySession = %Session;

If &MySession Then
 /* connection is good */

 &SearchQuery = &MySession.GetSearchQuery();
 &SearchQuery.IndexName = "PSC";
 &SearchQuery.Language = %Language;
 &SearchQuery.QueryText = SEARCH_RECORD.USER_QUESTION.Value;

 &Start = 1;
 &Size = 20;

 &SearchResultsCol = &SearchQuery.Execute(&Start, &Size);

  If &SearchResult.HitCount > 0 Then
   For &I = 1 to &SearchResultCol.Count
    &SearchResult = &SearchResultCol.Item(&I);

   /* Do processing */

   /* Do error checking */

     &ERRORCOL = &MySession.PSMessages;
     If (&ERRORCOL.Count <> 0) Then
       /* errors occurred . . . do processing */
     Else
      /* no errors */
     End-If;

   End-For;
  Else
  /* do processing for no returned matches to query */
```

```
  End-if;
 Else
 /* do processing for no connection */
End-If;
```

See Also

Chapter 29, Session Class , Error Handling , page 1560

Chapter 28, Search Classes , Parse, page 1518

Data Type of Search Objects

All search objects, like a SearchResult collection, a SearchField, and so on, are declared as type ApiObject. For example,

```
Local ApiObject &SearchResultCol;

Local ApiObject &SearchField;
```

Note. All Search objects can be declared only as Local.

Scope of the Search Objects

A search object can be instantiated only from PeopleCode using a PortalRegistry or Session object.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on. The limits placed on where you can use this object depend on how the object is used. For example, if you're doing search index administration, you are updating the database, and that can be done only in certain events.

Note. The Search API currently does not work in an Application Engine program run on the System 390.

Session Class Methods

A SearchQuery object must be instantiated from either a session object or a Portal Registry object. The Search API doesn't have any built-in functions.

In this section, we discuss the Session class methods. The methods are discussed in alphabetical order.

See Also

Chapter 29, Session Class , page 1559, Chapter 21, PortalRegistry Classes , page 1093.

GetSearchIndexes

Syntax

GetSearchIndexes()

Description

The **GetSearchIndexes** method returns a collection of SearchIndexes objects.

A SearchIndexes object and its children are used only for index administration, that is, creating and building search indexes. This method is *not* used with queries.

Parameters

None.

Returns

A SearchIndexes Collection object, populated with zero or more SearchIndex objects.

See Also

Chapter 28, Search Classes , SearchIndexes Collection, page 1527.

GetSearchQuery

Syntax

```
GetSearchQuery()
```

Description

The **GetSearchQuery** method returns an empty search query object used to start a search.

Parameters

None.

Returns

An empty search query object.

Example

```
&SearchQuery = %Session.GetSearchQuery();
```

See Also

Chapter 28, Search Classes , GetSearchQuery, page 1516.

PortalRegistry Class Methods

A SearchQuery object must be instantiated from either a session object or a Portal Registry object. The Search API doesn't have any built-in functions.

In this section, we discuss the PortalRegistry class methods. The methods are discussed in alphabetical order.

See Also

Chapter 23, ProcessRequest Class , page 1269.

BuildSearchIndex

Syntax

```
BuildSearchIndex(Language)
```

Description

The **BuildSearchIndex** method builds a portal search index for the specified language.

Search Collection Path Considerations

The Search Collection Path field in the configuration file points to a directory where collections can be built and queried through the search API. However, the **BuildSearchIndex** method *always* builds the collection in the default path, that is, in the following:

```
<PS_HOME>\data\search\<portal_name>\<db_name>\<lang_cd>
```

Parameters

Parameter	Description
Language	Specify the language code that you want the search index built in.

Returns

An optional Boolean value: True if the search index is built successfully, False otherwise.

GetSearchQuery

Syntax

```
GetSearchQuery()
```

Description

The **GetSearchQuery** method returns an empty search query object.

Note. When you use this method to get a search query, the index property is already set to the portal registry search index, and the language property is defaulted to the end-user's language.

Parameters

None.

Returns

An empty SearchQuery object with the index and language property already set.

Example

```
&Portal = %Session.GetPortalRegistry();

&Portal.Open("PORTAL");

&SearchQuery = &Portal.GetSearchQuery();
```

See Also

Chapter 28, Search Classes , GetSearchQuery, page 1515.

SearchQuery Class

The SearchQuery object is used to specify the search index and execute a query against it.

A SearchQuery object is returned by the following:

- The GetSearchQuery method from a PortalRegistry object.
- The GetSearchQuery method from a Session object.

See Chapter 28, Search Classes , GetSearchQuery, page 1516.

See Chapter 28, Search Classes , GetSearchQuery, page 1515.

SearchQuery Class Methods

In this section, we discuss the SearchQuery class methods.

Execute

Syntax

Execute(*Start*, *Size*)

Description

The **Execute** method runs a search query beginning at the document number specified by *Start*, and returns the number of results (or less) specified by *Size* in a SearchResult Collection.

Parameters

Parameter	Description
Start	Specify the document with which you want to start your query search. This parameter takes an integer, 1 or higher.
Size	Specify the number of results that you want returned. This parameter takes an integer value.

Returns

A SearchResult collection.

Example

```
&SearchResultCol = &SearchQuery.Execute(1, 20);
```

See Also

Chapter 28, Search Classes , SearchResult Collection, page 1522.

Parse

Syntax

Parse()

Description

Use the **Parse** method to verify the syntax of the query string specified through the QueryText property for syntax errors, such as whether there is a missing quotation mark or closing bracket.

The Parse method does not produce a collection of search results. If there are any errors, it returns a reference to a ParseResult collection that you can search through for errors. If there are not any errors, it retuns a Null object.

Parameters

None.

Returns

A reference to a ParseResult collection if there are errors, Null otherwise.

Example

```
Local ApiObject &MySession;
Local ApiObject &ParseResults;
Component ApiObject &SearchQuery, &SearchResultCol, &SearchResult;
Component Number &Start, &Size;

&MySession = %Session;

If &MySession Then
 /* connection is good */

  &SearchQuery = &MySession.GetSearchQuery();
  &SearchQuery.IndexName = "PSC";
  &SearchQuery.Language = %Language;
  &SearchQuery.QueryText = SEARCH_RECORD.USER_QUESTION.Value;

  &ParseResults = &SearchQuery.Parse();

  If All(&ParseResults) AND &ParseResults.ErrorCount > 0 OR ⇒
&ParseResults.WarningCound > 0 Then

   /* Process results */

  End-if;
End-if;
```

See Also

Chapter 28, Search Classes , ParseResult Collection, page 1520

SearchQuery Class Properties

In this section, we discuss the SearchQuery class properties. The properties are discussed in alphabetical order.

HitCount

Description

The **HitCount** property returns the total number of documents that actually match the query.

This property is read-only.

IndexName

Description

The **IndexName** property specifies the SearchIndex this query is to retrieve the result from.

You *must* set this property before a query can be performed on the SearchQuery object returned from the GetSearchQuery method.

This property is read-write.

Example

```
&SearchQuery.IndexName = "PSA";
```

KnowledgeBase

Description

Use the **KnowledgeBase** property to specify a topicset (that is, a file generated by using the mktopics Verity command for a search query. This property takes the name of a file, as a string.

This property is read-write.

Language

Description

The **Language** property specifies the language of the search index.

The default value for this property is the current user’s language.

This property is read-write.

ProcessedCount

Description

The **ProcessedCount** property returns the total number of documents that were subjected to the search query.

This property is read-only.

QueryText

Description

The **QueryText** property returns the query text to be submitted to the SearchQuery object.

This property is read-write.

ParseResult Collection

A ParseResult Collection is returned from the Parse SearchQuery method.

See Chapter 28, Search Classes , Parse, page 1518.

ParseResult Collection Methods

In this section, we discuss the ParseResult collection methods. The methods are discussed in alphabetical order.

First

Syntax

```
First()
```

Description

The **First** method returns the first ParseResult object in the ParseResult collection. If the ParseResult collection is empty, it returns Null.

Example

```
&MyResult = &MyParseResultCollection.First();
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next ParseResult object in the ParseResult collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

This method returns a Null when there are no more ParseResult objects in the collection.

Parameters

None.

Returns

Either a reference to a ParseResult object, or, if there are no more results in the ParseResult collection, a Null.

Example

```
Repeat
   /* &parseInfo.Message() contains the message, process it */
   &parseInfo.Next()
Until None(&parseInfo);
```

ParseResult Collection Properties

In this section, we discuss the ParseResult collection properties The properties are discussed in alphabetical order.

ErrorCount

Description

This property returns the number of ParseResult objects in this collection with a severity of ERROR.

This property is read-only.

Example

```
For &I = 1 to &ParseResults.ErrorCount
  /* process message */
End-For;
```

WarningCount

Description

This property returns the number of ParseResult objects in this collection with a severity of WARNING.

This property is read-only.

ParseResult Class

A ParseResult object is returned from either the First or Next ParseResult Collection methods.

See Chapter 28, Search Classes , ParseResult Collection, page 1520.

ParseResult Class Properties

In this section, we discuss the ParseResult class properties The properties are discussed in alphabetical order.

Message

Description

This property returns the text of the error or warning message as a string.

This property is read-only.

Severity

Description

This property returns the severity of the message as a string. Values are:

Value	Description
ERROR	Message is an error message.
WARNING	Message is a warning message.

This property is read-only.

SearchResult Collection

A SearchResult collection is returned from the Execute SearchQuery method.

See Chapter 28, Search Classes , Execute, page 1517.

SearchResult Collection Methods

In this section, we discuss the SearchResult collection methods. The methods are discussed in alphabetical order.

First

Syntax

First()

Description

The **First** method returns the first SearchResult object in the SearchResult collection. If the SearchResult collection is empty, it returns Null.

Example

```
&MyResult = &MyCollection.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns the SearchResult object with the position in the SearchResult collection specified by *number*.

Parameters

Parameter	Description
Number	Specify the position number in the collection of the SearchResult object that you want returned. Values start at 1.

Returns

A SearchResult object if successful, Null otherwise.

Example

```
For &I = 1 to &MyCollection.Count

  &MyResult = &MyCollection.Item(&I);

  /* Do processing */

End-For;
```

Next

Syntax

Next()

Description

The **Next** method returns the next SearchResult object in the SearchResult collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MyResult = &MyCollection.Next();
```

SearchResult Collection Property

In this section, we discuss the SearchResult collection property Count.

Count

Description

This property returns the number of SearchResult objects in the SearchResult collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

SearchResult Class

The SearchResult object represents a specific SearchResult in a SearchResult Collection object.

SearchResult objects are instantiated from a SearchResult Collection with the First, ItemByName or Next methods.

See Chapter 28, Search Classes , SearchResult Collection, page 1522.

SearchResult Class Properties

In this section, we discuss the SearchResult class properties. The properties are discussed in alphabetical order.

Key

Description

The **Key** property returns the key for this SearchResult object.

This property is read-only.

Score

Description

The **Score** property returns the score for this SearchResult object.

This property is read-only.

SearchFields

Description

The **SearchFields** property returns a SearchField Collection.

This property is read-only.

See Also

Chapter 28, Search Classes , SearchField Collection, page 1525.

SearchField Collection

A SearchField collection is returned by the SearchFields SearchResult property.

See Chapter 28, Search Classes , SearchFields, page 1525.

SearchField Collection Methods

In this section, we discuss the SearchField collection methods. The methods are discussed in alphabetical order.

First

Syntax

First()

Description

The **First** method returns the first SearchField object in the SearchField collection. If the SearchField collection is empty, it returns Null.

Example

```
&MyField = &MyCollection.First();
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the SearchField object specified by *Name*.

Parameters

Parameter	Description
Name	Specify the name of the SearchField you want returned.

Returns

A SearchField object if successful, Null otherwise.

Example

```
&MyField = &MyCollection.ItemByName(VALID_TO_DATE);
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next SearchField object in the SearchField collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MyField = &MyCollection.Next();
```

SearchField Collection Property

In this section, we discuss the SearchField collection property Count.

Count

Description

This property returns the number of SearchField objects in the SearchField collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

SearchField Class

A SearchField is returned by First, ItemByName, and Next SearchField Collection methods.

See Chapter 28, Search Classes , SearchField Collection, page 1525.

SearchField Class Properties

In this section, we discuss the SearchField class properties. The properties are discussed in alphabetical order.

Name

Description

The **Name** property returns the name of this SearchField object as a string.

This property is read-only.

Value

Description

The **Value** property returns the value for this SearchField object as a string.

This property is read-only.

SearchIndexes Collection

A SearchIndexes collection is returned by the GetSearchIndexes Session class property.

The SearchIndexes in this collection represent all of the *logical* search indexes that have been created in the PeopleSoft system.

Every search index built and used in a PeopleSoft system must have a logical search index associated with it.

See Chapter 28, Search Classes , GetSearchIndexes, page 1514.

SearchIndexes Collection Methods

In this section, we discuss the SearchIndexes collection methods. The methods are discussed in alphabetical order.

First

Syntax

First()

Description

The **First** method returns the first SearchIndex object in the SearchIndexes collection. If the SearchIndex collection is empty, it returns Null.

Example

```
&MySearchI = &MyCollection.First();
```

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the SearchIndex object specified by *Name*.

Parameters

Parameter	Description
Name	Specify the name of the SearchIndex that you want returned.

Returns

A SearchIndex object if successful, Null otherwise.

Example

```
&MySearchI = &MyCollection.ItemByName("MySearchIndex");
```

Next

Syntax

Next()

Description

The **Next** method returns the next SearchIndex object in the SearchIndexes collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MySearchI = &MyCollection.Next();
```

SearchIndex Collection Property

In this section, we discuss the SearchIndex collection property Count.

Count

Description

This property returns the number of SearchIndex objects in the SearchIndexes collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

SearchIndex Class

A SearchIndex is returned by First, ItemByName, and Next SearchIndexes Collection methods.

A SearchIndex represents a logical search index, and all the information describing it, in the PeopleSoft system. Every search index built and used in a PeopleSoft system must have a logical search index associated with it.

See Chapter 28, Search Classes , SearchIndexes Collection, page 1527.

SearchIndex Class Method

In this section, we discuss the SearchIndex class method Save.

Save

Syntax

Save()

Description

Use the **Save** method to save any changes to the search index.

Note. If you change a value of one of the properties of the subobjects (such as some of the search options, and so on) the change won't be written to the database until the SearchIndex object has been saved.

Parameters

None.

Returns

A Boolean value: True if the SearchIndex object is saved successfully, False otherwise.

SearchIndex Class Properties

In this section, we discuss the SearchIndex class properties. The properties are discussed in alphabetical order.

Note. If you change a value of one of the properties, the change won't take effect until the SearchIndex object has been saved.

ExtraOptions

Description

This property is used for the spidering search indexes. Parameters that are not covered in the SearchIndex object and its children can be passed to the Verity spidering utility using this property.

This property is read-write.

FSOpts

Description

This property returns a reference to a Search FS Options object, for FSYS type search indexes.

This property is read-only.

See Also

Chapter 28, Search Classes , Search FS Options Class, page 1549.

HTTPOpts

Description

This property returns a reference to a Search HTTP Options object, for HTTP type search indexes.

This property is read-only.

See Also

Chapter 28, Search Classes , Search HTTP Options Class, page 1546.

Languages

Description

This property returns a reference to the Search Language collection.

This property is read-only.

See Also
Chapter 28, Search Classes , Search Language Collection, page 1539.

Location

Description

This property specifies the file system location where the search index (collection) is located. You must use a full path name.

This property is read-write.

Name

Description

This property returns the name of the SearchIndex, as a string.

This property is read-only.

RecOpts

Description

This property returns a reference to a SearchRecord Options collection, for RECD type search indexes.

This property is read-only.

See Also
Chapter 28, Search Classes , SearchRecord Options Class, page 1532.

Schedules

Description

This property returns a reference to the Search Schedule collection.

This property is read-only.

See Also
Chapter 28, Search Classes , Search Schedule Collection, page 1542.

Type

Description

The **Type** property specifies the type of index. This property takes a string value. The values are:

Value	Description
RECD	An index generated from a PeopleSoft record
HTTP	An index generated from a spider search of the World Wide Web (WWW)

Value	Description
FSYS	An index generated from a spider search of a file system
PRTL	An index generated from a portal
USER	A user defined index (that is, not one of the previous indexes)

This property is read-write.

SearchRecord Options Class

A reference to a search record options object is returned by the RecOpts SearchIndex property.

This object is valid only for Search Index types of RECD. These types of indexes are used to index data in a PeopleSoft database. The properties used with this object represent the values that must be set for this type of search index.

See Chapter 28, Search Classes , RecOpts, page 1531.

SearchRecord Options Class Properties

In this section, we discuss the SearchRecord option class properties. The properties are discussed in alphabetical order.

Note. If you change a value of one of the properties, the change won't take effect until the SearchIndex object has been saved.

Fields

Description

This property returns a reference to a SearchRecordField collection.

The SearchRecordField collection describes the records and fields that are indexed.

The maximum number of fields returned as part of this collection is 50.

This property is read-only.

See Also

Chapter 28, Search Classes , SearchRecordField Collection, page 1535.

Filter

Description

This property specifies a filter (that is, a SQL WHERE clause) used to filter out unwanted records from being indexed, as a string.

This property is read-write.

IncrementalView

Description

This property specifies a view used for accessing the records that are to be used for incremental indexing, as a string.

This property is read-write.

RecName

Description

This property returns the record name for the primary record that the search index is based on. Multiple records can be used, but they must all share a common key structure. The record name returned with this property specifies this common base key structure.

This property is read-only.

VeggieKey

Description

This property specifies a replacement pattern for the VdkVgwKey (VeggieKey) used by Verity to uniquely identify a document. The setting of this property determines what is filled in for the unique identifier for each document that gets indexed. Text in this string is replaced with either literal text in the unique identifier or a value determined by one of the following tags:

Tag	Description
<pairs/>	Inserts a string of NAME=VALUE; pairs for each key on the Record being indexed
<row/>	Inserts the Record keys in a SQL-like syntax

Tag	Description
<field fieldname='*MYFIELD*'/>	Inserts the value of *MYFIELD* for the Row being indexed (if it exists in the record; an error occurs when you save the data if it is missing)
<sql stmt='*SQL STATEMENT*'/>	Inserts the value returned by the SQL statement. Only the first row returned is accepted. SQL statements that return more than one column cause an error at runtime.

To use these tags, you include them as part of the VeggieKey string. The indexing process replaces them with values. For example, given the following Row of data:

```
Record: PROD_SRCH
Field: PRODUCTID=A11111 (a key on the record)
Field: DESCR="Baby goldfish"
Field: PRICE=5.00
```

You could produce any of the following unique identifiers with various settings of VeggieKey:

VeggieKey Setting	String Produced
"<pairs/>"	"PRODUCTID=A11111;"
"SELECT * FROM <row/>"	"SELECT * FROM PS_PROD_SRCH WHERE PRODUCTID='A11111'"
"<pairs/> <field filedname='DESCR'/>"	"PRODUCTID=A11111; Baby goldfish"
"Total price: <sql stmt='SELECT SUM(PRICE) FROM PS_PROD_SRCH'/>"	"Total price: 550.78"

Note that the last example is not truly unique, although it is legal. It is up to the application to make sure that the setting of VeggieKey produces a unique string for each indexed document, as the indexing process does not perform this check. Applications that use VdkVgwKey to identify the results of search queries cannot find the correct document if the documents do not have a unique key.

This property is read-only.

ZoneOptions

Description

This property specifies the zone option for the index as a string. Values are:

Value	Description
ONE	All the fields for each row of data are inserted into the same zone in the word index. This is a simple scheme which does not allow you to search only parts of the word index but may be appropriate for applications without easily-differentiated text regions.
FLD	One zone is created for each field. The data in that field for each row is inserted into the same zone. This enables applications to search only a specific part of the word index at a time, with a potential performance gain. This does not happen automatically, however; the query interface built by the application must support searching in specific zones. The name of the zone created is the name of the field. The build process may take longer with this option set.

This property is read-only.

SearchRecordField Collection

A SearchRecordField collection is returned from the Fields SearchRecord Options property.

The SearchRecordField collection describes the records and fields that are indexed.

The maximum number of fields returned as part of this collection is 50.

See Chapter 28, Search Classes , Fields, page 1532.

SearchRecordField Collection Methods

In this section, we discuss the SearchRecordField collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*Record, Field*)

Description

Use the **DeleteItem** method to delete the SearchRecordField specified by the record and field name.

This method is executed immediately, however, the values are updated in the database only when the SearchIndex is saved.

Parameters

Parameter	Description
Record	Specify a record name as a string.
Field	Specify a field name as a string.

Returns

A Boolean value: True, item was deleted successfully, False otherwise.

First

Syntax

```
First()
```

Description

The **First** method returns the first SearchRecordField object in the SearchRecordField collection. If the SearchRecordField collection is empty, it returns Null.

Example

```
&MySearchField = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(Record, Field)
```

Description

Use the **InsertItem** method to insert a SearchRecordField object into the SearchRecordField collection.

This method is executed immediately, however, the values are updated in the database only when the SearchIndex is saved.

If the item you are trying to insert already exists, this method returns Null.

Parameters

Parameter	Description
Record	Specify a record name as a string.
Field	Specify a field name as a string.

Returns

A reference to a new SearchRecordField object if successful, Null otherwise.

ItemByName

Syntax

ItemByName(*Record, Field*)

Description

The **ItemByName** method returns the SearchRecordField object specified by the record and field name.

Parameters

Parameter	Description
Record	Specify a record name as a string.
Field	Specify a field name as a string.

Returns

A SearchRecordField object if successful, Null otherwise.

Example

```
&MySearchField = &MyCollection.ItemByName("SEARCH", "DATE");
```

Next

Syntax

Next()

Description

The **Next** method returns the next SearchRecordField object in the SearchRecordField collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MySearchOpts = &MyCollection.Next();
```

SearchRecordField Collection Property

In this section, we discuss the SearchRecordField collection property Count.

Count

Description

This property returns the number of SearchRecordField objects in the SearchRecordField collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

SearchRecordField Class

A SearchRecordField is returned by the First, ItemByName, InsertItem, and Next SearchRecordField collection methods.

A SearchRecordField object describes the records and fields that are indexed.

See Chapter 28, Search Classes , SearchRecordField Collection, page 1535.

SearchRecordField Class Properties

In this section, we discuss the SearchRecordField class properties. The properties are discussed in alphabetical order.

Note. If you change a value of one of the properties, the change won't take effect until the SearchIndex object has been saved.

FieldName

Description

This property returns the field name for this SearchRecordField as a string.

This property is read-only.

IsAttachment

Description

This property specifies whether the search record field is an attachment. This property takes a Boolean value: True, the field is an attachment, False otherwise.

This property is read-write.

IsVerityField

Description

This property specifies whether the search record field is a Verity field. This property takes a Boolean value: True, the field is a Verity field, False otherwise.

This property is read-write.

IsWordIndex

Description

This property specifies whether the search record field is a word index. This property takes a Boolean value: True, the field is a word index, False otherwise.

This property is read-write.

RecordName

Description

This property returns the name of the record as a string.

This property is read-only.

Search Language Collection

A reference to a Search Language collection is returned by the Languages property.

See Chapter 28, Search Classes , Languages, page 1530.

Search Language Collection Methods

In this section, we discuss the Search Language collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*Name*)

Description

Use the **DeleteItem** method to delete the Search Language specified by *Name*.

This method is executed immediately, however, the values are updated in the database only when the SearchIndex is saved.

Parameters

Parameter	Description
Name	Specify the name of the Search Language that you want to delete.

Returns

A Boolean value: True, item was deleted successfully, False otherwise.

First

Syntax

```
First()
```

Description

The **First** method returns the first Search Language object in the Search Language collection. If the Search Language collection is empty, returns Null.

Example

```
&MyLang = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(LanguageCode, MapLanguageCode)
```

Description

Use the **InsertItem** method to insert a Search Language object into the Search Language collection.

This method is executed immediately, however, the values are updated in the database only when the SearchIndex is saved.

If the item you are trying to insert already exists, this method returns Null.

Parameters

Parameter	Description
LanguageCode	Specify a language code as a string.
MapLanguageCode	Specify a language code to map *LanguageCode* to (typically this is the same as *LanguageCode*).

Returns

A reference to a new Search Language object if successful, Null if not successful.

ItemByName

Syntax

```
ItemByName(LanguageCode)
```

Description

The **ItemByName** method returns the Search Language object specified by *LanguageCode*.

Parameters

Parameter	Description
LanguageCode	Specify the language code of the object that you want to retrieve.

Returns

A reference to a Search Language object if successful, Null otherwise.

Example

```
&MyLang = &MyCollection.ItemByName("ENG");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next Search Language object in the Search Language collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MyLang = &MyCollection.Next();
```

Search Language Collection Property

In this section, we discuss the Search Language collection property Count.

Count

Description

This property returns the number of Search Language objects in the Search Language collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Search Language Class

A reference to a Search Language object is returned by the First, ItemByName, InsertItem, and Next Search Language Collection methods.

See Chapter 28, Search Classes , Search Language Collection, page 1539.

Search Language Class Properties

In this section, we discuss the Search Language class properties. The properties are discussed in alphabetical order.

LanguageCd

Description

This property specifies the language code for the Search Language object as a string.

This property is read-only.

MapLanguageCd

Description

Specify a language code to map the given language to (typically this is the same as LanguageCode).

This property is read-only.

Search Schedule Collection

A reference to a Search Schedule collection is returned by the Schedules SearchIndex property.

See Chapter 28, Search Classes , Schedules, page 1531.

Search Schedule Collection Methods

In this section, we discuss the Search Schedule collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*Name*)

Description

Use the **DeleteItem** method to delete the Search Schedule specified by *Name*.

This method is not executed automatically. It is executed only when the SearchIndex is saved.

Parameters

Parameter	Description
Name	Specify the name of the Search Schedule that you want to delete.

Returns

A Boolean value: True, item was deleted successfully, False otherwise.

First

Syntax

```
First()
```

Description

The **First** method returns the first Search Schedule object in the Search Schedule collection. If the Search Schedule collection is empty, this method returns Null.

Example

```
&MySch = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(RunCntrlID)
```

Description

Use the **InsertItem** method to insert a Search Schedule object into the Search Schedule collection.

This method is not executed automatically. It is executed only when the SearchIndex is saved.

If the item you are trying to insert already exists, this method returns Null.

Parameters

Parameter	Description
RunCntrlId	Specify the run control ID of the search schedule object as a string.

Returns

A reference to a new Search Schedule object if successful, Null if not successful.

ItemByName

Syntax

```
ItemByName(RunCntrlId)
```

Description

The **ItemByName** method returns the Search Schedule object specified by *RunCntrlId*.

Parameters

Parameter	Description
RunCntrlId	Specify the run control ID of the search schedule object as a string.

Returns

A Search Schedule object if successful, Null otherwise.

Example

```
&MySchedule = &MyCollection.ItemByName("MySched");
```

Next

Syntax

```
Next()
```

Description

The **Next** method returns the next Search Schedule object in the Search Schedule collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MySched = &MyCollection.Next();
```

Search Schedule Collection Property

In this section, we discuss the Search Schedule collection property Count.

Count

Description

This property returns the number of Search Schedule objects in the Search Schedule collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Search Schedule Class

A reference to a Search Schedule object is returned by the First, ItemByName, InsertItem, and Next Search Schedule Collection methods.

See Chapter 28, Search Classes , Search Schedule Collection, page 1542.

Search Schedule Class Properties

In this section, we discuss the Search Schedule class properties. The properties are discussed in alphabetical order.

Note. If you change a value of one of the properties, the change won't take effect until the SearchIndex object has been saved.

BuildType

Description

This property specifies how you want the index rebuilt with this schedule item. Values are:

Value	Description
INCR	Incremental updates an existing index. Use this schedule item for indexes with small amounts of data, and for cleaning up large indexes periodically when they start to fragment.
RBLD	Rebuild deletes the index and starts from scratch. Use this to add to an existing, large index.

This property is read-write.

RunCntlId

Description

This property returns the Run Control ID associated with the schedule item.

This property is read-only.

RunRecurrence

Description

This property specifies the name of the Recurrence Definition as a string. For example: "Daily Search Rebuild".

This property is read-write.

ServerName

Description

This is the name of the Process Scheduler Server that processes the requests for this schedule item, for example, "PSNT".

This property is read-write.

Search HTTP Options Class

A reference to a Search HTTP Options object is returned by the HTTPOpts SearchIndex property.

This object is valid only for Search Index types of HTTP. These types of indexes spider URLs. The properties available using this object represent values that must be set for spidering URL's.

See Chapter 28, Search Classes , HTTPOpts, page 1530.

Search HTTP Options Class Properties

In this section, we discuss the Search HTTP Option class properties. The properties are discussed in alphabetical order.

Note. If you change a value of one of the properties, the change won't take effect until the SearchIndex object has been saved.

AllowHTTPS

Description

This property is used to cause the HTTP Spider Gateway to descend HTTPS (SSL) links and HTTP links when it builds the index. This property takes a Boolean value: True, use the HTTPS (SSL) links, False otherwise.

Use of this feature requires a separate license obtained directly from Verity.

This property is read-write.

DomainLimit

Description

This property limits spidering (indexing) to the specified domain. This property takes a string value.

This property is read-write.

GlobList

Description

This property specifies a list of filename "globs" which the indexing process uses to filter its list. The GlobListType determines how the filtering is done. The value of this property is a space-separated list of filenames with or without wildcards. The following is an example of the value for this property:

```
"*.doc *.txt *.html robots.txt"
```

This property is read-write.

See Also

Chapter 28, Search Classes , GlobListType, page 1547.

GlobListType

Description

This property determines how the values specified with GlobList are used. Values are:

Value	Description
ALL	Ignore the values specified with GlobList. Index all filenames encountered regardless of their pattern.
EXC	Exclude the values specified with GlobList. Index everything except names that match a pattern in GlobList.
INC	Include the values specified with GlobList. Index only filenames that match a pattern in GlobList and exclude all other filenames.

This property is read-write.

See Also

Chapter 28, Search Classes , GlobList, page 1547.

LinkDepth

Description

This property determines how many links the HTTP Spider descends from its starting point as a number.

A value of 1, for example, indexes the original page as well as the documents linked to by that page, and then stop.

A value of 2 indexes the original page, and documents linked to it, as well as documents linked to by the second page.

PeopleSoft recommends not setting this value over 10 without combining it with the DomainLimit property as the amount of data retrieved into the index increases geometrically with the LinkDepth.

This property is read-write.

See Also
Chapter 28, Search Classes , DomainLimit, page 1547.

MIMEList

Description

This property is used in a similar fashion to GlobList. It is a space-separated list of MIME Types. The MIMEListType determines how they affect indexing. The following is an example of a value for this property:

```
"text/html text/plain application/*"
```

This property is read-write.

See Also
Chapter 28, Search Classes , MIMEListType, page 1548, Chapter 28, Search Classes , GlobList, page 1547.

MIMEListType

Description

This property is used like GlobListType, but applies to MIMEList instead of GlobList. Instead of testing the pattern of the filename, this property tests the MIME-type detected for the document when determining whether to index it.

This property is read-write.

See Also
Chapter 28, Search Classes , MIMEList, page 1548, Chapter 28, Search Classes , GlobListType, page 1547.

ProxyHost

Description

This property specifies the hostname or IP address of the HTTP proxy, if the HTTP Spider Gateway needs an HTTP proxy to connect to remote websites during the indexing process.

This property is read-write.

ProxyPort

Description

This property specifies the port number of the HTTP proxy set in ProxyHost as a number.

This property is read-write.

StartOpts

Description

This property returns a reference to a Search Start Options collection.

This property is read-only.

See Also

Chapter 28, Search Classes , Search Start Options Collection, page 1551.

Search FS Options Class

A reference to a Search FS Options object is returned by the FSOpts SearchIndex property.

This object is valid only for Search Index types of FSYS. These types of indexes spider a file system. The properties available using this object represent the values that must be set for spidering a file system.

See Chapter 28, Search Classes , FSOpts, page 1530.

Search FS Options Class Properties

In this section, we discuss the Search FS Option class properties. The properties are discussed in alphabetical order.

Note. If you change a value of one of the properties, the change won't take effect until the SearchIndex object has been saved.

GlobList

Description

This property specifies a list of filename "globs" which the indexing process uses to filter its list. The GlobListType determines how they affect indexing. The value of this property is a space-separated list of filenames with or without wildcards. The following is an example of the value for this property:

```
"*.doc *.txt *.html robots.txt"
```

This property is read-write.

See Also

Chapter 28, Search Classes , GlobListType, page 1550.

GlobListType

Description

This property determines how the values specified with GlobList are used. Values are:

Value	Description
ALL	Ignore the values specified with GlobList. Index all filenames encountered regardless of their pattern.
EXC	Exclude the values specified with GlobList. Index everything except names that match a pattern in GlobList.
INC	Include the values specified with GlobList. Index only filenames that match a pattern in GlobList and exclude all other filenames.

This property is read-write.

See Also

Chapter 28, Search Classes , GlobList, page 1549.

MIMEList

Description

This property is used like GlobList. It is a space-separated list of MIME Types. The MIMEListType determines how they affect indexing. The following is an example of the value for this property:

```
"text/html text/plain application/*"
```

This property is read-write.

See Also

Chapter 28, Search Classes , MIMEListType, page 1550, Chapter 28, Search Classes , GlobList, page 1549.

MIMEListType

Description

This property is used like GlobListType, but applies to MIMEList instead of GlobList. Instead of testing the pattern of the filename, this property tests the MIME-type detected for the document when determining whether to index it.

This property is read-write.

See Also

Chapter 28, Search Classes , MIMEList, page 1550, Chapter 28, Search Classes , GlobListType, page 1550.

StartOpts

Description

This property returns a reference to a Search Start Options collection.

This property is read-only.

See Also

Chapter 28, Search Classes , Search Start Options Collection, page 1551.

Search Start Options Collection

The values in the Search Start Options Collection get passed to the **−start** parameter of the Verity spider utility. These values represent the starting points for spidering.

A Search Start Option collection is returned from:

- StartOpts Search HTTP Options class property
- StartOpts Search FS Options class property

See Search HTTP Options Class property

See Chapter 28, Search Classes , StartOpts, page 1549.

See Chapter 28, Search Classes , StartOpts, page 1549.

Search Start Options Collection Methods

In this section, we discuss the Search Start Options collection methods. The methods are discussed in alphabetical order.

DeleteItem

Syntax

DeleteItem(*Value*)

Description

Use the **DeleteItem** method to delete the Search Start Options specified by *Value*.

This method is executed immediately, however, the values are updated in the database only when the SearchIndex is saved.

Parameters

Parameter	Description
Value	Specify the value of the Search Start Option that you want to delete.

Returns

A Boolean value: True, item was deleted successfully, False otherwise.

First

Syntax

```
First()
```

Description

The **First** method returns the first Search Start Options object in the Search Start Options collection. If the Search Start Options collection is empty, this method returns Null.

Example

```
&MySearchOpts = &MyCollection.First();
```

InsertItem

Syntax

```
InsertItem(Value)
```

Description

Use the **InsertItem** method to insert a Search Start Options object into the Search Start Options collection.

This method is executed immediately, however, the values are updated in the database only when the SearchIndex is saved.

If the item you are trying to insert already exists, this method returns Null.

Parameters

Parameter	Description
Value	Specify the value of the Search Start Option that you want to insert.

Returns

A reference to a new Search Start Option object if successful, Null if not successful.

ItemByName

Syntax

```
ItemByName(Value)
```

Description

The **ItemByName** method returns the Search Start Options object specified by *Value*.

Parameters

Parameter	Description
Value	Specify the value of the Search Start Options that you want returned.

Returns

A Search Start Options object if successful, Null otherwise.

Example

```
&MySearchOpts = &MyCollection.ItemByName("MyStartOptions");
```

Next

Syntax

Next()

Description

The **Next** method returns the next Search Start Options object in the Search Start Options collection. You can use this method only after you have used the **First** method: otherwise the system doesn't know where to start.

Example

```
&MySearchOpts = &MyCollection.Next();
```

Search Start Options Collection Property

In this section, we discuss the Search Start Options collection property Count.

Count

Description

This property returns the number of Search Start Options objects in the Search Start Options collection, as a number.

This property is read-only.

Example

```
&COUNT = &MY_COLLECTION.Count;
```

Search Start Options Class

A reference to a Search Start object is returned by the First, ItemByName, InsertItem, and Next Search Start Options Collection methods.

See Chapter 28, Search Classes , Search Start Options Collection, page 1551.

Search Start Options Class Properties

In this section, we discuss the Search Start Options class properties. The properties are discussed in alphabetical order.

Note. If you change a value of one of the properties, the change won't take effect until the SearchIndex object has been saved.

IsDomainRestricted

Description

The value of this property is passed to the **−domain** option of the Verity spider utility. This property specifies whether the domain is restricted. This property takes a Boolean value: True, the domain is restricted, False otherwise.

This property is read-write.

IsHostRestricted

Description

This property specifies whether the host is restricted. This property takes a Boolean value: True, the host is restricted, False otherwise.

This property is read-write.

Value

Description

This property returns the value of the Search Start Options, as a string.

This property is read-only.

Search API Examples

The following PeopleCode programs are examples of how to use the Search API.

General Purpose Search API Example

The following is a general example.

```
/*---------------------------------------------------------------------
  Get a search query object.
---------------------------------------------------------------------*/
&SearchQuery = %Session.GetSearchQuery();
```

```
/*---------------------------------------------------------------------
  Set the search index to access
---------------------------------------------------------------------*/
&SearchQuery.IndexName = "PSA"

/*---------------------------------------------------------------------
  Set the language to access
---------------------------------------------------------------------*/
&SearchQuery.Language = %Language;

/*---------------------------------------------------------------------
  Set the text of the query the user entered
---------------------------------------------------------------------*/
&SearchQuery.QueryText = "Java AND C++ AND (’MIT’ in EDUCATION)"

/*---------------------------------------------------------------------
  Execute the search passing in the starting position and "chunk size" (get me the
  first 20 result or the third  20)
---------------------------------------------------------------------*/
&SearchResultsColl = &SearchQuery.Execute(1, 20);
/*---------------------------------------------------------------------
 Get the first result.
---------------------------------------------------------------------*/
&SearchResult = &SearchResultsColl.First();

While (&SearchResult <> Null)

   /*---------------------------------------------------------------------
     Example of getting the key.  In this case assume that the key contains a file⇒
 path to a resume.
   ---------------------------------------------------------------------*/
   &ResumePath = &SearchResult.Key;

   /*---------------------------------------------------------------------
     Example of getting the value of a Field.
   ---------------------------------------------------------------------*/
   &ApplicantIdSearchField = &SearchResult.SearchFields.ItemByName("APPLICANT_ID")
   &ApplicantId = &URLSearchField.Value

   /*---------------------------------------------------------------------
     Another example of a field using dot notation to simplify the code.
   ---------------------------------------------------------------------*/
   &Address = &SearchResult.SearchFields.ItemByName("ADDRESS").Value

   /*---------------------------------------------------------------------
     Call a function to do something with the search result.
   ---------------------------------------------------------------------*/
   DoSomethingUseful(&ResumePath, &ApplicantId, &Address);

   /*---------------------------------------------------------------------
```

```
      Get the next result of the search.
    ---------------------------------------------------------------------------*/
    &SearchResult = &SearchResultsColl.Next();
End-While;
```

Portal Search API Example

The following is a Portal search example.

```
&SearchKey = %Request.GetParameter("SEARCH_TEXT");
&StartPosition = %Request.GetParameter("START_POSITION")
SearchResultChunkSize = 1000;

/*---------------------------------------------------------------------------
  Get a portal registry object.
---------------------------------------------------------------------------*/
&Portal = %Session.GetPortalRegistry();

/*---------------------------------------------------------------------------
  Open the desired portal.
---------------------------------------------------------------------------*/
&Portal.Open("PORTAL")

/*---------------------------------------------------------------------------
  Get a search query object.
---------------------------------------------------------------------------*/
&SearchQuery = &PORTAL.GetSearchQuery();

/*---------------------------------------------------------------------------
  Set the text of the query the user entered
---------------------------------------------------------------------------*/
&SearchQuery.QueryText = &SearchKey

/*---------------------------------------------------------------------------
  Execute the search passing in the starting position and "chunk size" (get me the⇒
 first 20 result or the third  20)
---------------------------------------------------------------------------*/
&SearchResultsColl = &SearchQuery.Execute(&StartPosition, &SearchResultChunkSize);
/*---------------------------------------------------------------------------
 Get the first result.
---------------------------------------------------------------------------*/
&SearchResult = &SearchResultsColl.First();

While (&SearchResult <> Null)

   /*---------------------------------------------------------------------------
     Example of getting the key.  In the portal case, the key is the URL, but in⇒
 the general case it could be a product id or some kind of other database key.
   ---------------------------------------------------------------------------*/
```

```
   &SearchKey = &SearchResult.Key;

   /*---------------------------------------------------------------------
 Example of getting the Field.  In the portal case, the field is the URL.  Yes, it⇒
 is redundant with the key but the key includes some {} around the URL and this⇒
 just makes it easier to get the URL.
   ---------------------------------------------------------------------*/
   &URLSearchField = &SearchResult.SearchFields.ItemByName("URL")
   &URL = &URLSearchField.Value

   /*---------------------------------------------------------------------
     Another example of a field using dot notation to simplify the code.
   ---------------------------------------------------------------------*/
   &ContentProvider = &SearchResult.SearchFields.ItemByName("Content⇒
Provider").Value

   /*---------------------------------------------------------------------
     Call a function to insert the result into the page.
   ---------------------------------------------------------------------*/
   AddStuffToPage(&URL, &ContentProvider);

   /*---------------------------------------------------------------------
     Get the next result of the search.
   ---------------------------------------------------------------------*/
   &SearchResult = &SearchResultsColl.Next();

End-While;
&Portal.Close();
```

CHAPTER 29

Session Class

This chapter provides an overview of Session class and discusses the following topics:

- Error handling.
- Regional settings.
- Trace settings.
- Session object declaration.
- Scope of a Session object.
- Session class built-in function.
- Session class reference section.

Understanding Session Class

PeopleTools provides a family of APIs for external access into the PeopleSoft system. The Session class is the root of the APIs. It controls access to the PeopleSoft system, controls the environment, and enables you to do error handling for all APIs from a central location.

The PeopleSoft APIs are intended to be used, and are supported, in the following environments.

- PeopleSoft Windows client with either an existing two-tier connection or an existing connection to an application server.
- PeopleSoft Application Engine batch program.
- Non-PeopleSoft program connecting to an application server.

The PeopleSoft APIs are exposed to the following language environments:

- PeopleCode
- OLE/COM
- C/C++
- Java

The APIs that are accessible using a session object are:

- Component Interface Classes
- PortalRegistry Classes
- Query Classes

- Search Class
- Tree Classes

Note. Tree Classes are accessible only through PeopleCode.

A session object controls the following:

- Security and access to the PeopleSoft system.
- Errors and error handling .
- Regional settings (that is, internationalization) for language, dates, times, and numbers.
- Tracing.

See Also

Chapter 10, Component Interface Classes , page 773

Chapter 21, PortalRegistry Classes , page 1093

Chapter 24, Query Classes , page 1303

Chapter 33, Tree Classes , page 1653

Security and Access to the PeopleSoft System

A session object controls security and access to the PeopleSoft system. You must "sign-on" to the PeopleSoft session with a valid user ID and password when you use the **Connect** method. In addition, there's an external authentication parameter that you can use with the connect.

Note. Additional authentication will be available after this release.

Error Handling

The session object handles error processing for all the APIs, such as Component Interfaces, the Query API, and so on. From the session object, you can check if there have been any errors.

Note. The exception to this is Subscription PeopleCode. All errors for Subscription PeopleCode get logged to the application message error table.

Error messages include system error messages or messages from the message catalog (a common set of error messages used by all PeopleSoft applications.) For a Component Interface, the session object level errors also contain any text errors that may have occurred from running ExecuteEdits.

On the session object, you can use the following properties to initially check for errors:

- **ErrorPending** indicates whether there are API errors
- **WarningPending** indicates whether there are API warnings

All errors are contained in the **PSMessages** collection. (The **PSMessages** property on a Session object returns this collection.)

Each item in this collection is a **PSMessage object**. A PSMessage object contains information about the specific error that has occurred, such as the explain text for the error, the message set number, and so on. (The type of information depends on the type of error.)

If the error was caused by a Component Interface, a contextual string is attached to the end of the **Text** property of the **PSMessage** object. This contextual string contains the exact location of the error, that is, which field in which data collection, on which row, caused the error. This string is also accessible (by itself) using the **Source** property.

When errors are loaded into the PSMessages collection depends on the type of error. System errors (that is, errors in the connection to the PeopleSoft Session) are logged as they occur. When an API error is logged depends on the API, as some APIs can be run in either interactive mode (meaning errors are logged as they happen) or in non-interactive mode (so errors are logged only when a particular method is run.)

If you are using Visual Basic or another COM environment, the PeopleSoft API system raises a COM exception the first time **ErrorPending** changes from False to True. It will *not* raise another exception until the PSMessages collection is cleared, which sets **ErrorPending** back to False.

Note. If you've called an API from an Application Engine program, all errors are also logged in the Application Engine error log tables.

See Also

Chapter 29, Session Class , Accessing a PSMessages Collection , page 1570

Displaying Error Messages

Use the following PeopleCode to display messages from the PSMessage collection:

```
Local ApiObject &Session, &PSMessages;
   &Session = %Session;
   &PSMessages = &Session.PSmessages;
   If (&Session <> Null ) Then
       If &PSMessages.Count > 0 Then
               For &i = 1 To &PSMessages.Count
                  &MsgSetNbr = &PSMessages.Item(&i).MessageSetNumber;
                  &MsgNbr = &PSMessages.Item(&i).MessageNumber;
                  MessageBox( 0 , " ", &MsgSetNbr, &MsgNbr, "Message not found");
               End-For;
         &PSMessages.DeleteAll();
            End-If;
End-If;
```

Regional Settings

Regional settings enable the user to set the display of numbers, dates, times, and currencies to comply with usage in a specific locale. The session object exposes some of these values to the API through the **RegionalSettings** property. This property returns a regional settings object that has properties you can use to change these settings.

See Also

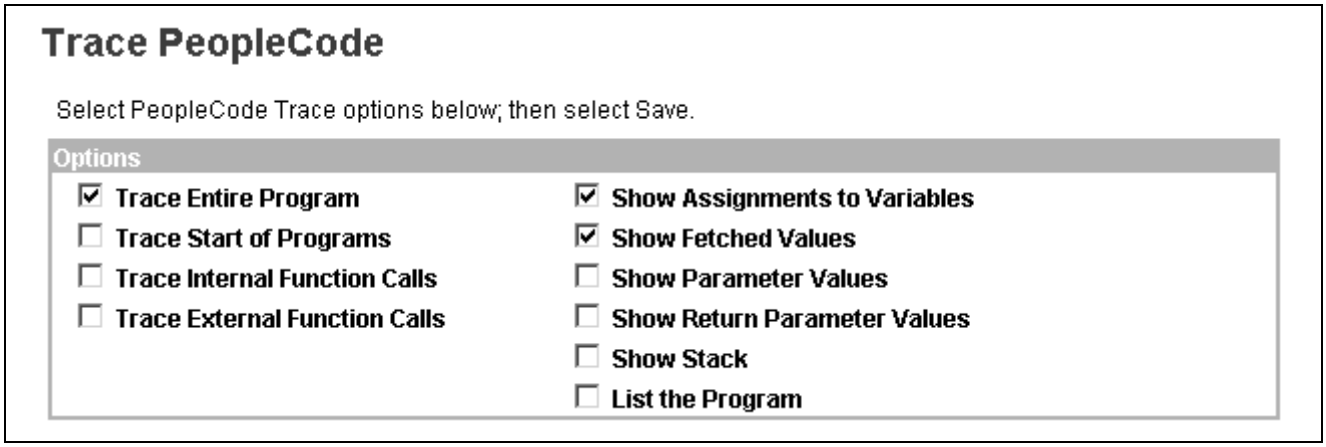
Chapter 29, Session Class , Regional Settings Class Properties, page 1577

Trace Settings

The PeopleSoft debug pages enables you to control the environment in which your PeopleTools session is running. These pages include:

- Trace PeopleCode
- Trace SQL
- Trace Page

These pages enable you to select the SQL trace options and the PeopleCode trace options that you want. The **TraceSettings** object (assessable from a session object) lets you control many of the same options during your session.



Trace PeopleCode page

Not every option on the debug pages is represented with a trace settings property.

See Also

PeopleSoft Server Tools Administration, PeopleTools Utilities, Debug Utilities

Session Object Declaration

All session objects are declared as type ApiObject. For example,

```
Local ApiObject &MYSESSION;
Global ApiObject &PSMessage;
```

The following session objects can be declared as Global or Component:

- Session
- PSMessages Collection
- PSMessage

All other session objects *must* be declared as Local.

In this section, we discuss the following topics:

- State considerations.
- Considerations for declaring conditions.

State Considerations

In the PeopleSoft Internet Architecture, all processing occurs on the server. If you do something in your PeopleCode program that causes a trip back to the user before the end of your PeopleCode program, the PeopleCode program *cannot*regain its state, that is, it can't reset local variables, and so on. This applies to all APIs that use the Session object.

If you must go back to the user before the end of your PeopleCode program, you must set all your objects to NULL before you go, or else you receive a runtime error.

For example, the following code causes an error:

```
&Session = %Session;
Rem &Session = Null;
WinMessage("Hello");
```

The following example does *not* cause an error:

```
&Session = %Session;
&Session = Null;
WinMessage("Hello");
```

Considerations for Declaring Collections

You *must* declare all collections for C/C++ and COM as objects, or else you receive errors at runtime.

For example if you're using RegionalSettings, include the following your declarations for a VB program:

```
Dim oRegionalSettings As Object
```

Then, use the following code to populate this variable:

```
Set oRegionalSettings = oCISession.RegionalSettings
```

```
oRegionalSettings.LanguageCd = "GER"
```

Scope of a Session Object

A Session can be instantiated from PeopleCode, from a Visual Basic program, from C++, COM, and so on.

See the example section in Connect for more details.

This object can be used anywhere you have PeopleCode, that is, in Application Engine PeopleCode, record field PeopleCode, and so on.

Session Class Built-in Function

Chapter 3, System Variables , %Session , page 577

Chapter 1, PeopleCode Built-in Functions, GetSession , page 269

Session Class Methods

In this section, we discuss the Session class methods. The methods are discussed in alphabetical order.

Connect

Syntax

```
(version, {"EXISTING" | //PSoftApplicationServer:JoltPort}, UserID, Password, ExtAuth)
```

Description

The **Connect** method connects a session object to a PeopleSoft application server.

Note. This function remains for backward compatibility only. Use the **%Session** system variable to return a reference of a session object instead.

If you already have a PeopleSoft session running (you are already connected to a PeopleSoft application server and are running a component, and so on) you must specify EXISTING, and not the *//PSoftApplicationServer:JoltPort*. Typically, use the parameter value of EXISTING from PeopleCode, not from other language environments.

If you are using an existing connection to the application server, you *cannot*specify a different user ID or password. If you don't specify these values as NULL string, you must specify the exact same user ID (and password) as the one that originally started the session.

Parameters

Parameter	Description
version	Specify the API version that the client is expecting. Future releases will use this parameter. For now, you must use a 1.
EXISTING | *//PSoftApplicationServer: JoltPort*	Specify **EXISTING** if you're currently connected to an application server. Otherwise, specify the named application server machine and the IP port to connect to on an application server machine.
UserID	Specify the PeopleSoft user ID to use for the connection. This must be a valid user ID. If you are using an existing connection, you can specify a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.
Password	Specify the password associated with the user ID to use for the connection. This must match the password assigned for this user ID *exactly*. If you are using an existing connection, you can specify a NULL string (that is, two quotation marks with no blank space between them ("")) for this parameter.
ExtAuth	This parameter is required, but it is unused in this release. You must enter a 0 for this parameter.

Returns

A Boolean value: True if the connection completed successfully, False otherwise.

Example

The following is an example of the code you would use if there is an existing connection to the application server:

```
Local ApiObject &MYSESSION;

&MYSESSION = GetSession();
&MYSESSION.Connect(1, "EXISTING", "", "", 0);
```

The following is an example connecting to a PeopleSoft system using a Visual Basic program. It first checks for error messages. If there are any errors, the text of the error is displayed to the user.

```
On Error GoTo eMessage

    Dim oSession As Object
    Dim oPSMessage As Object

    Dim oBC As Object

    Set oSession = CreateObject("PeopleSoft.Session")

    oSession.Connect 1, "//PSSOFTO061998:7000", "PTDMO", "PTDMO", 0

    ‘Application Specific Processing

eMessage:
    If (oSession.PSMessages.Count > 0) Then
```

```
        For i = 1 To oSession.PSMessages.Count
           Set oPSMessage = oSession.PSMessages.Item(i)
           MsgBox (oPSMessage.Text)
        Next i
    End If
```

See Also

Chapter 3, System Variables , %Session , page 577.

Disconnect

Syntax

Disconnect()

Description

The **Disconnect** method disconnects the session associated with the session object executing the method from that PeopleSoft session.

This method returns a Boolean value: True if successfully disconnected, False otherwise.

API Instantiation Methods

The following is a list of the other methods used with the session object to instantiate an API object such as a Component Interface, a tree structure, and so on.

Component Interface:

- FindCompIntfcs(*partial_name*): returns a collection of Component Interfaces based on the partial Component Interface name
- GetCompIntfc(**COMPINTFC**.*componentname)*: returns a Component Interface object

Portal Registry:

- FindPortalRegistries(*partial_name*): returns a collection of PortalRegistry objects
- GetLocalNode(): returns a node object
- GetNodes(): returns a collection of node objects
- GetPortalRegistry(): returns a PortalRegistry object
- GetRemoteNodes(): returns a collection of node objects

Query:

- FindQueryDBRecords():returns a reference to a QueryDBRecord collection, filled with zero or more records
- FindQueries():returns a reference to a Query collection, filled with zero or more queries
- FindQueriesDateRange(*StartDateString*, *EndDateString*): returns a reference to a Query collection, filled with zero or more queries that match the specified date range

- GetQuery(): returns a query object
- GetQuerySecurityProfile(): a QuerySecurityProfile object
- SearchQueryDBRecord(*SearchType*, *Pattern*, *CaseSensitive*): returns a QueryDBRecord collection
- SearchPrivateQueries(*QueryType*, *SearchType*, *Pattern*, *CaseSensitive*): a query collection
- SearchPublicQueries(*QueryType*, *SearchType*, *Pattern*, *CaseSensitive*): a query collection

Search:

- GetSearchIndexes(): returns a SearchIndexes collection
- GetSearchQuery(): returns a search query object

Tree:

- FindTree(*SetId*, *UserKeyValue*, *TreeName*, *EffDt*, *BranchName*): returns a collection of tree objects based on the partial values set for the parameters
- FindTreeStructure(*StructureID*): returns a collection of tree structure objects based on the partial tree structure name
- GetTree(*SetId*, *UserKeyValue*, *TreeName*, *EffDt*, *BranchName*): returns a tree object
- GetTreeStructure(*StructureID*): returns a tree structure object

Session Class Properties

This section describes the Session class properties. The properties are discussed in alphabetical order.

ErrorPending

Description

This property indicates whether there are any error messages pending for the API that is currently running. After this property has been set, it returns True until the PSMessages collection is cleared (or deleted.)

This property is read-only.

Example

```
If &MYSESSION.ErrorPending Then
   &COUNT = &MYSESSION.PSMessages.Count;
   For &I = 1 To &COUNT
      &ERROR = &MYSESSION.PSMessages.Item(&I);
      &TEXT = &ERROR.Text;
      WinMessage("Error text " | &TEXT);
   End-For;

&MYSESSION.PSMessages.DeleteAll();
```

PSMessages

Description

This property returns a reference to a PSMessages collection object. If there are no messages, this property returns an object reference to PSMessages with a count of zero.

This property is read-only.

Example

```
If &SESSION.PSMessages.Count > NULL Then
  /* there are messages do processing */
End-if;
```

See Also

Chapter 29, Session Class , PSMessages Collection Methods , page 1570.

PSMessagesMode

Description

This property is used to determine how messages are output. This property takes either a numeric value or a constant. The default value is 1 (%PSMessages_CollectionOnly).

You must set this property before you check the type of message, that is, you can't check the type of message, then decide how it's displayed.

This property sets the value for the session. You can change modes during a session, such as, if you're starting a Component Interface. However, after you run the Component Interface, you should set the value back. For example:

```
&OldMode = &Session.PSMessageMode;
&Session.PSMessagesMode = 1;
...
&Session.PSMessagesMode = &OldMode;
```

You can specify either a numeric value or a constant. The values are:

Numeric Value	Constant Value	Description
0	%PSMessages_None	None.
1	%PSMessages_CollectionOnly	PSMessage Collection only (default).
2	%PSMessages_MessageBoxOnly	Message box only.
3	%PSMessages_Both	Both collection and message box.

Note. If you set this property to 0 (%PSMessages_None), *all* messages are ignored. Use this option with caution.

This property is read-write.

Example

```
Local ApiObject &SESSION;

&SESSION = %Session;
&MMODE = &SESSION.PSMessagesMode;
&SESSION.PSMessagesMode = %PSMessages_MessageBoxOnly;
```

RegionalSettings

Description

This property returns a reference to a RegionalSettings collection object.

This property is read-only.

See Also

Chapter 29, Session Class , Regional Settings Class Properties, page 1577.

Repository

Description

This property returns a reference to a Repository object.

See Also

PeopleSoft Integration Tools and Utilities, The PeopleSoft API Repository

SuspendFormatting

Description

This property turns off the PeopleSoft automatic formatting when coercing a string into a field. Formatting is typically necessary for external Component Interface users (such as for a Visual Basic program,) yet should be suspended for a PeopleCode program. You should not typically need to set this value yourself, as it's set appropriately based on where the session object is created.

This property takes a Boolean value: True means formatting is suspended, False means it isn't suspended. If you start your session from a PeopleCode program, the default for this property is True. If you don't start the session from a PeopleCode program, the default is False.

This property is read-write.

TraceSettings

Description

This property returns a reference to a TraceSettings collection object.

This property is read-only.

See Also

Chapter 29, Session Class , Trace Setting Class Properties , page 1580.

WarningPending

Description

This property indicates whether there are any warning messages pending for the API that is currently running.

This property is read-only.

Accessing a PSMessages Collection

Use the PSMessages collection to return all of the messages that occur during the session. The following example finds all the messages listed in the PSMessages collections when the Component Interface Save methods fails. The example assumes that &CI is the Component Interface object reference.

```
If Not &CI.Save() Then
  /* save didn't complete */
  &COLL = &MYSESSION.PSMessages;
  For &I = 1 to &COLL.Count
   &ERROR = &COLL.Item(&I);
   &TEXT = &ERROR.Text;

   /* do message processing */

  End-For;
&COLL.DeleteAll(); /* Delete all messages in queue */
End-if;
```

As you evaluate each message, delete it from the PSMessages collection. Or, delete all the messages in the queue at once using DeleteAll.

If there are multiple errors when saving a Component Interface, all errors are logged to the PSMessages collection, not just the first occurrence of an error.

Considerations for Declaring Collections

You *must* declare all collections for C/C++ and COM as objects, or else you receive errors at runtime.

PSMessages Collection Methods

This section describes the PSMessages collection methods. The methods are discussed in alphabetical order.

DeleteAll

Syntax

```
DeleteAll()
```

Description

The **DeleteAll** method deletes all the PSMessage objects in the PSMessages collection executing the method. You should use this property after you have processed all the errors in the collection. This method also resets the status of ErrorPending to False.

Returns

This method returns a Boolean value: True if all PSMessages in the collection were successfully deleted, False otherwise.

See Also

Chapter 29, Session Class , ErrorPending, page 1567.

DeleteItem

Syntax

```
DeleteItem(number)
```

Description

The **DeleteItem** method deletes the PSMessage object that exists at the *number* position in the PSMessages collection executing the method.

Parameters

Parameter	Description
number	Specify the position number in the collection of the PSMessage object that you want to delete. All other items within the collection are renumbered.

Returns

This method returns a Boolean value: True if the PSMessage object was successfully deleted, False otherwise.

Example

```
&ERROR.DeleteItem(&I);
```

First

Syntax

```
First()
```

Description

The **First** method returns the first PSMessage object in the PSMessages collection object executing the method.

Example

```
&ERROR = &PSMESSAGE.First();
```

Item

Syntax

Item(*number*)

Description

The **Item** method returns a PSMessage object that exists at the *number* position in the PSMessages collection executing the method

Parameters

Parameter	Description
number	Specify the position number in the collection of the PSMessage object that you want returned.

Returns

A reference to a PSMessage object, NULL otherwise.

Example

```
&PSMSGCOL = &SESSION.PSMessages;
For &I = 1 to &PSMSGCOL.Count;
  &PSMESSAGE = &PSMSGCOL.Item(&I);
  /* do error processing */
End-For;
```

Next

Syntax

Next()

Description

The **Next** method returns the next PSMessage object in the PSMessages collection object executing the method. You can use this method only after you have used either the **First** or **Item** methods: otherwise the system doesn't know where to start.

PSMessages Collection Property

This section describes the PSMessages collection property.

Count

Description

This property returns the number of PSMessage objects in the PSMessages collection object.

This property is read-only.

Example

```
&COUNT = &PSMESSAGE.Count;
```

PSMessage Class Properties

This section describes the PSMessages class properties. The properties are discussed in alphabetical order.

ExplainText

Description

This property returns the explanation text (as a string) for the error message associated with the PSMessage object. You can use this property in conjunction with the **MessageNumber** property (which contains the error message number) and the **MessageSetNumber** property (which contains the error message set number.)

Note. ExplainText should be accessed only when necessary because it requires a second database read or application server roundtrip.

This property is read-only.

MessageNumber

Description

This property returns the error message number (as a number) for the error message associated with the PSMessage object. You can use this property in conjunction with the **MessageSetNumber** property (which contains the error message set number) and the **ExplainText** property (which contains text explaining the error.)

This property is read-only.

MessageSetNumber

Description

This property returns the error message set number (as a number) for the error message associated with the PSMessage object. You can use this property in conjunction with the **MessageNumber** property (which contains the error message number) and the **ExplainText** property (which contains text explaining the error.)

This property is read-only.

MessageType

Description

This property indicates the type of the message, whether it's from the message catalog, if it's a warning, error, or information error.

This property returns a numeric value. The values are:

Value	Description
0	Message is not a message catalog entry or there is no message.
1	Message has a severity of Error.
2	Message has a severity of Warning.
3	Message has a severity of Information.

This property is read-only.

Source

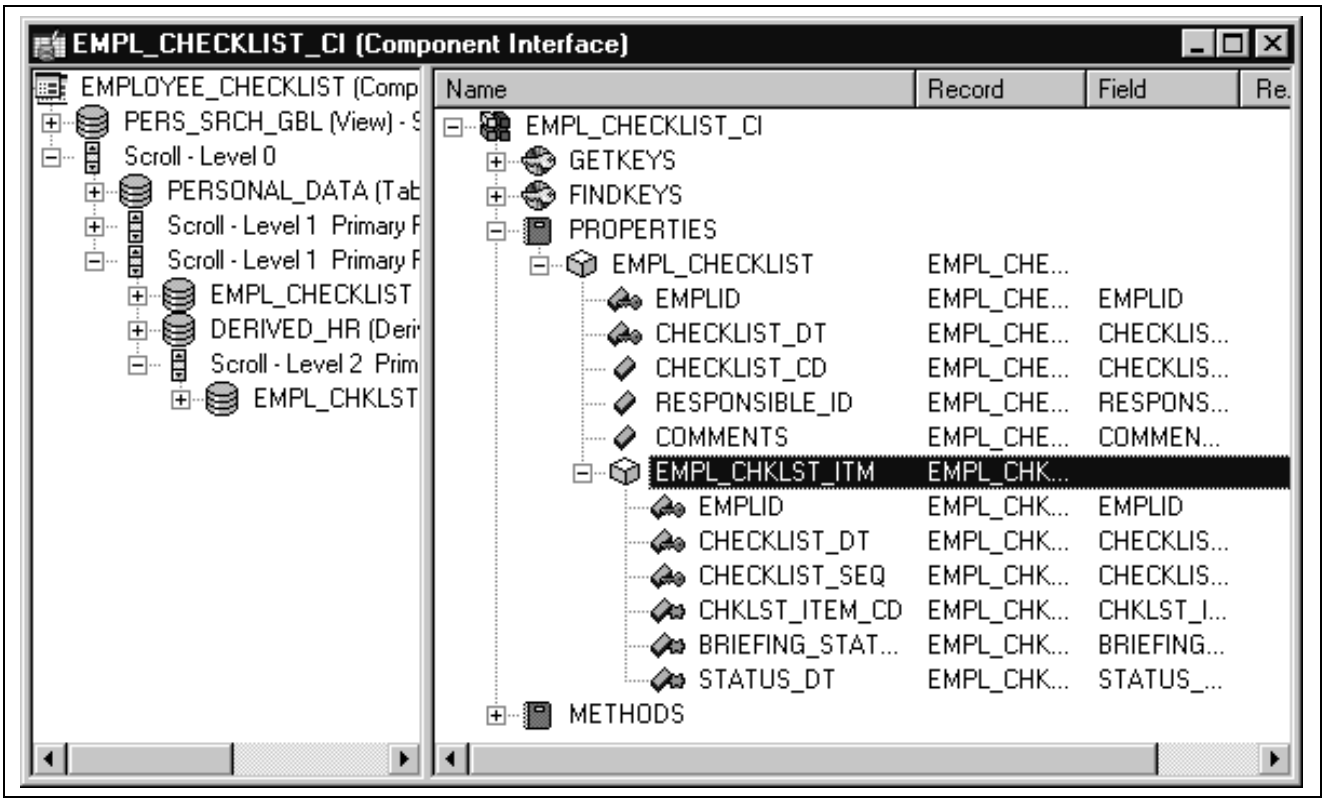
Description

This property returns a string indicating the actual field that's in error. The syntax of the string that's returned is as follows:

```
ComponentInterfaceName.[CollectionName(Row).[CollectionName(Row).⇒
[CollectionName(Row)]]].PropertyName
```

This string is also returned as part of the **Text** property.

For example, suppose you had a Component Interface named EMPL_CHKLST_CI.



Sample Component Interface

The following indicates that the first level field, EMPLID, has the error:

```
EMPL_CHKLST_CI.EMPLID
```

The Component Interface EMPL_CHKLST_BC has a data collection (scroll) called EMPL_CHKLST_ITM. Suppose that it has 3 rows, and the STATUS_DT field was in error. The **Source** property would return the following string:

```
EMPL_CHKLST_CI.EMPL_CHKLST_ITM(3).STATUS_DT
```

You can use the **Source** property in conjunction with the **MessageNumber** property (which contains the error message number), the **MessageSetNumber** property (which contains the error message set number), the **ExplainText** property (which contains text explaining the error), and the **Text** property (which contains the text of the message.)

This property is read-only.

Example

The following example code finds the error messages and displays them to the user. It finds the field that caused the error and displays that also.

```
Local ApiObject &PSMESSAGE;
Local Boolean &FIND;
Local string &SOURCE, &FIELD, &TEXT;
Local number &START, &LEN, &NSTART, &FLEN;
.
.
```

```
.
For &I = 1 to &SESSION.PSMessages.Count;
  &PSMESSAGE = &SESSION.PSMessages.Item(&I);
  /* only display errors, not warnings, to user */
  If &PSMESSAGE.Type = 1 Then
    &TEXT = &PSMESSAGE.Text;
  /* find name of field in error */
    &SOURCE = &PSMESSAGE.Source;
    &LEN = Len(&SOURCE);
  /* find last dot before field */
    &FIND = False;
    &START = Find(".", &SOURCE);
    While Not (&FIND)
        &NSTART = Find(".", &SOURCE, &START + 1);
          If &NSTART = 0 Then
          &FIND = True;
          Else
              &START = &NSTART;
          End-If;
    End-While;
    &FLEN = &LEN - &START;
    &FIELD = Substring(&SOURCE, &START + 1, &FLEN);
   /* display text and field to user */
   Winmessage("You received the following error: " | &TEXT | "For field " |⇒
 &FIELD);
End-For;
```

Text

Description

This property returns the text of the message (as a string) for the error message associated with the PSMessage object. It also includes a contextual string that contains the name of the field that generated the error. The contextual string has the following syntax:

```
{ComponentInterfaceName.[CollectionName(Row).[CollectionName(Row).⇒
[CollectionName(Row)]]].PropertyName}
```

The contextual string (by itself) is available using the **Source** property of the PSMessage.

You can use the **Text** property in conjunction with the **MessageNumber** property (which contains the error message number), the **MessageSetNumber** property (which contains the error message set number), and the **ExplainText** property (which contains text explaining the error.)

This property is read-only.

Example

```
Local ApiObject &MYSESSION, &COL, &ERROR;
Local string &TEXT;
.
```

```
.
.
If &MYSESSION.ErrorPending Then
   /* received error */
   &COLL = &MYSESSION.PSMessages;
   For &I = 1 to &COLL.Count
     &ERROR = &COLL.Item(&I);
     &TEXT = &ERROR.Text;
     /* do error processing */
   End-For;
End-if;
```

See Also

Chapter 29, Session Class , Source, page 1574.

Regional Settings Class Properties

This section describes the regional settings class properties. The properties are discussed in alphabetical order.

ClientTimeZone

Description

This property specifies the client time zone. This property takes a string value.

This property is read-write.

CurrencyFormat

Description

This property specifies how currency values are displayed. This property takes a number value. The default value is 1.

In the following values, @ represents the character specified with the CurrencySymbol property.

Value	Description
0	@1.1
1	1.1@
2	@.1.1
3	1.1.@

This property is read-write.

CurrencySymbol

Description

This property specifies the currency symbol. This property takes a string value. The default value is "$".

This property is read-write.

DateFormat

Description

This property specifies the date format defaults. PeopleTools supports the Short Date Format specification (MDY, DMY, or YMD), and the Date separator setting. When you add a Date field in a record definition, you indicate whether you want to display the century. PeopleTools Date and DateTime fields always show leading zeros for month, day, and year.

This property takes a number value. The values for this property are:

Value	Description
0	MDY (default)
1	DMY
2	YMD
3	Default value on system

This property is read-write.

DateSeparator

Description

This property specifies the character used to separate the month, day, year in the date. This property takes a string value. The default value is "/".

This property is read-write.

DecimalSymbol

Description

This property specifies the character used to separate the fractional of the number from the whole. This property takes a string value. The default value for this property is ".".

Note. If you change the value of this property to a comma (,), you should adjust the DigitGroupingSymbol to a different character or some applications may not operate correctly.

This property is read-write.

DigitGroupingSymbol

Description

This property specifies the character used as a thousand or other grouping separator. This property takes a string value. The default value for this property is ",".

This property is read-write.

LanguageCD

Description

This property specifies the language to be used for the transaction. This property takes a string value. The default value for this property is the user's base language.

This property is read-write.

Note. If you change this value dynamically using PeopleCode, this value will *not* change back to the original value when you disconnect. You must manually set it back to the original value.

Example

The following example sets the language code, then at the end of the program, sets it back to the original value.

```
Local ApiObject &SESSION;
Local ApiObject &CI;

/** Get Session ApiObject **/
&SESSION = %Session;

/** Get Message ApiObject **/
&MSG = GetMessage();
&RS = &MSG.GetRowset();

/** Get Component Interface **/
&CI = &SESSION.GetCompIntfc(compIntfc.NAMES_CI);

&REG_LNG = &SESSION.RegionalSettings.LanguageCd;

For &TRANS_NUM = 1 To &RS.RowCount

/** Store Language Code in Temp Var **/
&LNG = &RS(&TRANS_NUM).PSCAMA.LANGUAGE_CD.Value;

/** Set the Language for this Transaction **/
If &LNG <> &REG_LNG Then
&SESSION.RegionalSettings.LanguageCd = &LNG;
End-If;
/* application specific processing */

&SESSION.RegionalSettings.LanguageCd = &REG_LNG;
```

```
&SESSION.Disconnect();
End-For
```

UseLocalTime

Description

This property specifies whether the local time zone of the client should be used to format time fields. This property takes a Boolean value. The default value is True.

This property is read-write.

1159Separator

Description

This property specifies the character used to separate hours/minutes/seconds for time represented on a 12-hour clock (that is, 11:00 p.m.). This property takes a string value. The default is "."

This property is read-write.

2359Separator

Description

This property specifies the character used to separate hours/minutes/seconds for time represented as a 24-hour clock (that is, 23:00.) This property takes a string value. The default is "."

This property is read-write.

Trace Setting Class Properties

Each of the following properties relates to the debug pages in PeopleTools.

This section describes the trace setting class properties. The properties are discussed in alphabetical order.

See Also

PeopleSoft Server Tools Administration, PeopleTools Utilities, Debug Utilities

API

Description

This property is part of the PeopleSoft API trace and sets the **API Information** value.

This property takes a Boolean value: True means the trace will be run, False means it won’t.

This property is read-write.

COBOLStmtTimings

Description

This property is part of the SQL trace and sets the **COBOL Statement Timings** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

ConnDisRollbackCommit

Description

This property is part of the SQL trace and sets the **Connect, disconnect, rollback,** and **commit** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

DBSpecificCalls

Description

This property is part of the SQL trace and sets the **Database API-specific Calls** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

ManagerInfo

Description

This property is part of the SQL trace and sets the **Manager Information** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

NetworkServices

Description

This property is part of the SQL trace and sets the **Network Services** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

NonSSBs

Description

This property is part of the SQL trace and sets the **All Other API Calls besides SSBs** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

OutputUNICODE

Description

This property specifies if the trace file is written in ANSI or UNICODE (for example, **Output Unicode Trace File**).

This property takes a Boolean value: True means the file will be written in UNICODE, False means it will be ANSI.

This property is read-write.

PCExtFcnCalls

Description

This property is part of the PeopleCode trace and sets the **Trace External Function Calls** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCFcnReturnValues

Description

This property is part of the PeopleCode trace and sets the **Show Function Return Value** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCFetchedValues

Description

This property is part of the PeopleCode trace and sets the **Show Fetched Values** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCIntFcnCalls

Description

This property is part of the PeopleCode trace and sets the **Trace Internal Function Calls** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCListProgram

Description

This property is part of the PeopleCode trace and sets the **List the Program** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCParameterValues

Description

This property is part of the PeopleCode trace and sets the **Show Parameter Values** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCProgramStatements

Description

This property is part of the PeopleCode trace and sets the **Trace Each Statement in Program** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCStack

Description

This property is part of the PeopleCode trace and sets the **Show Stack** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCStartOfPrograms

Description

This property is part of the PeopleCode trace and sets the **Trace Start of Programs** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCTraceProgram

Description

This property is part of the PeopleCode trace and sets the **Trace Evaluator Instructions** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

PCVariableAssignments

Description

This property is part of the PeopleCode trace and sets the **Show Assignments to Variables** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

RowFetch

Description

This property is part of the SQL trace and sets the **Row Fetch** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

SQLStatement

Description

This property is part of the SQL trace and sets the **SQL Statement** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

SQLStatementVariables

Description

This property is part of the SQL trace and sets the **SQL Statement Variables** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

SSBs

Description

This property is part of the SQL trace and sets the **Set Select Buffers** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

SybBindInfo

Description

This property is part of the SQL trace and sets the **Sybase Bind Information** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

SybFetchInfo

Description

This property is part of the SQL trace and sets the **Sybase Fetch Information** value.

This property takes a Boolean value: True means the trace will be run, False means it won't.

This property is read-write.

TraceFile

Description

This property specifies the location of the trace file (that is, **PeopleTools Trace File** value.)

This property takes a string value.

This property is read-write.

CHAPTER 30

SOAPDoc Class

This chapter provides an overview of SOAPDoc class and discusses the following topics:

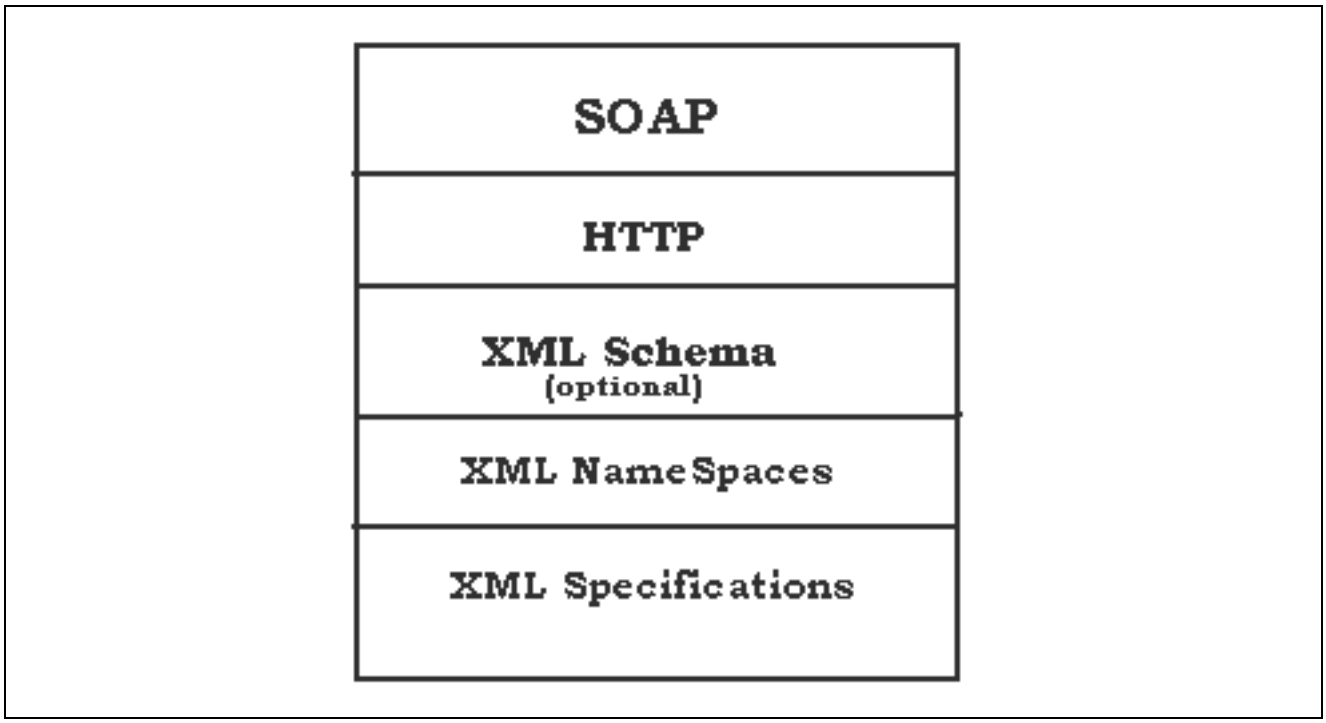
- SOAP message exchange model.
- SOAP message document.
- The SOAPDoc class.
- SOAPDoc object creation.
- The ValidateSOAPDoc method.
- SOAPDoc objects and methods.
- Data type of a SOAPDoc object.
- Scope of a SOAPDoc object.
- SOAPDoc built-in functions.
- SOAPDoc class methods.
- SOAPDoc class properties.

Understanding SOAPDoc Class

The Simple Object Access Protocol (SOAP) is a lightweight protocol for defining synchronous messages in a distributed Web environment. It's an XML based protocol that can be used in combination with other protocols such as HTTP using URLs as endpoints for a message. SOAP is an application of XML and XML namespaces and optionally uses XML Schemas.

SOAP is an industry standard, defined by the World Wide Web Consortium (W3C), that defines an XML grammar for identifying a method name and the method parameters and for returning the results.

More specific information about SOAP can be found in the W3C specification on SOAP. The SOAP serialization rules can also be found in the complete specifications and aren't outlined in this chapter. Also included in the W3C specification are sections on using SOAP with HTTP, which is also outside the domain of this chapter. You can also send and receive HTTP messages through the Integration Gateway.



Soap is an application built with XML and HTTP technologies

See Also

http://www.w3.org

PeopleSoft Integration Broker, Sending and Receiving Messages

SOAP Message Exchange Model

SOAP messages are always synchronous. They can implement Request/Response synchronous messages with the response having the output parameters returned in another SOAP document to the sender.

SOAP supports HTTP-POST and HTTP-GET formats. PeopleSoft supports only HTTP, using the Integration Gateway.

Processing the message requires validation of the mandatory sections of the SOAP XML document and understanding SOAP namespaces.

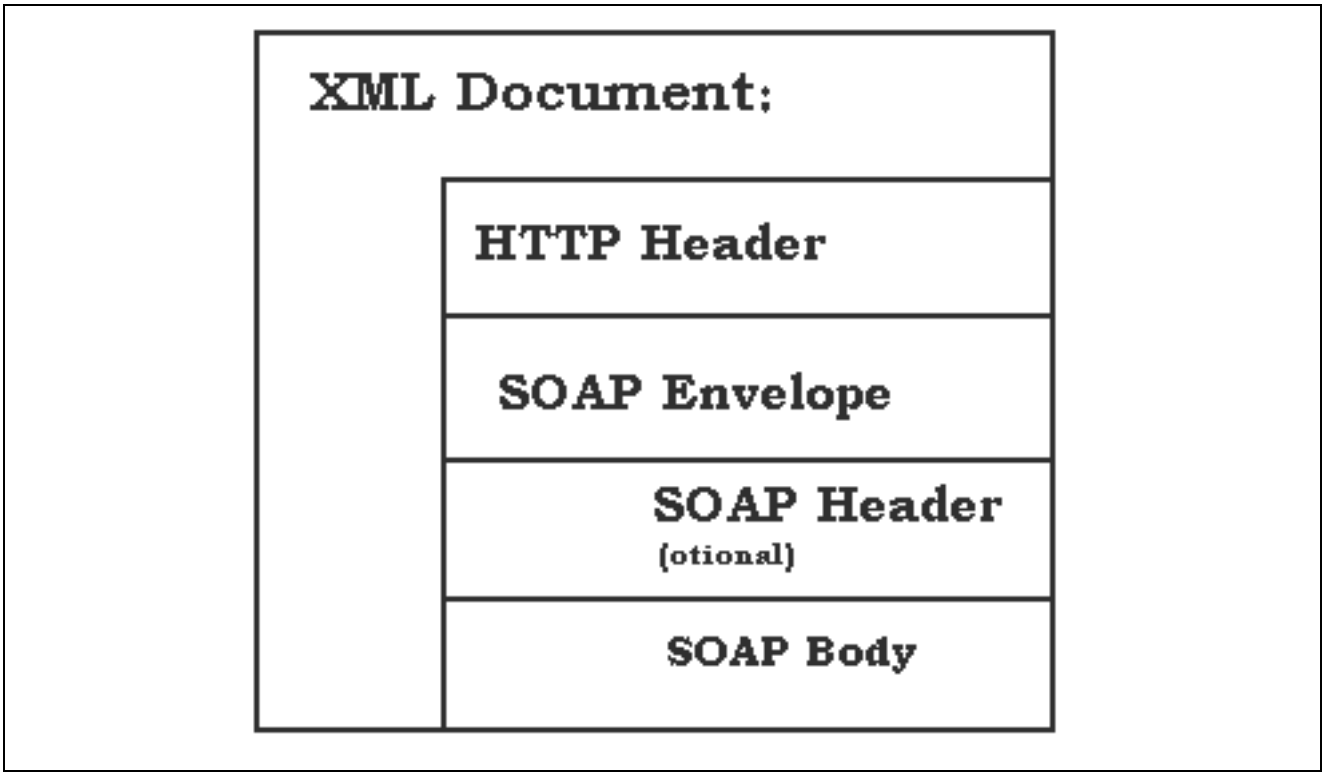
SOAP Message Document

A SOAP Message is an ordinary XML document that consists of the following parts:

- A mandatory SOAP envelope.
- An optional SOAP header.

• A mandatory SOAP body.

SOAP Message Part	Description
SOAP Envelope	Is the top element of the XML document representing the message. It defines the content of the message, as well as the framework of what is in a message, how to process it, who should deal with it and whether it is optional or mandatory.
SOAP Header	Contains header information, and attributes that can be set to encode and further identify the type of processing as well as additional features of the message. This portion is optional.
SOAP Body	Contains the call and response information intended for the recipient of the message. A SOAP Fault element can appear in the body of the SOAP message to carry error or status information.



SOAP parts in an XML document

The following are *required*for a SOAP message:

• Encoded using XML.

• Have a SOAP envelope.

• Have a SOAP body.

- Use the SOAP encoding and envelope namespaces.

The following are optional for a SOAP message:

- SOAP Header.
- XML Schema use.

The following are *invalid* for a SOAP message:

- DTD references.
- XML Processing Instructions.
- XML CData Sections.

The SOAPDoc Class

The SOAPDoc class is a separate class in PeopleCode that works similarly to the XmlDoc and XmlNode classes. Many of the same methods and properties that work with an XmlDoc or XmlNode work with a SOAPDoc. However, there are also methods and properties that don't apply to a SOAPDoc.

The following XmlDoc and XmlNode methods do *not* apply to a SOAPDoc:

- AddCDataSection
- AddProcessInstruction
- CreateDocumentType
- InsertCDataSection
- InsertProcessInstruction

If a Merchant Profile has been created it can be used with the SOAPDoc class and the Message class. You can use the **XmlDoc** property to convert the message object to an SOAPDoc object, then use the SOAPDoc methods and properties, as well as the XmlDoc methods and properties that are applicable.

You can only use the message object when the XmlDoc object is structured.

SOAPDoc Object Creation

There are a few steps to creating a SOAPDoc. The following is an overview of these steps. More detail about the methods and properties mentioned in this section can be found in the reference section of this chapter.

SOAP Header Section

Because the SOAP header is optional, the header is not validated when you use the **ValidateSOAPDoc** method.

A SOAP header must be added *before* any other parts of the SOAPdoc.

Use the **AddHeader** method to add a header to a SOAPDoc.

Use the **GetHeaderNode** method to access an existing SOAPDoc header.

After you have a reference to the header, you can use XmlDoc methods to read the contents of the header (like any node.)

See Also

Chapter 30, SOAPDoc Class , AddHeader, page 1598

Chapter 30, SOAPDoc Class , GetHeaderNode, page 1602

SOAP Envelope Section

A SOAP envelope is a required portion of the XML Message and is generated automatically using the SOAP schema as specified by SOAP W3C.

A SOAP envelope must be added *before* you add a SOAP body or a SOAP method. This is an optional method. You need to use this method only if you need to set any additional attributes for the Envelope section. The default SOAP schemas are added automatically.

To set additional custom attributes or schemas on the Envelope element, use the **AddEnvelope** method. This method adds the envelope element tags and the default schemas for either SOAP or UDDI depending on the value of the parameter passed in. The default encoding styles are also set here. Only one envelope section can be set within one PeopleSoft SOAP XmlDoc.

The SOAPDoc adds only a default URL to the envelope. For some UDDI sites, you may need to add a specific schema yourself. To do this, generate the envelope with no schema, then access the envelope section and add the specific schema URL using the AddAttribute method.

See Also

Chapter 30, SOAPDoc Class , AddEnvelope, page 1596

SOAP Body Section

The body is the XmlDoc where the method is defined. The method **AddBody** starts the body section of the XML SOAPDoc and must be in the XmlDoc before adding in method sections of the message. Only one body section can be set within one SOAPDoc. This is an optional method and must called only if you need to set any additional attributes for the body section. Otherwise the body section is automatically generated in the XmlDoc.

The SOAP body must be added *after* you add the SOAP envelope, and *before* you add any methods.

See Also

Chapter 30, SOAPDoc Class , AddBody, page 1595

SOAP Method Section

The **AddMethod** method adds the name of the method element being processed. Additional attributes for this method can be set using the **AddAttribute** and **InsertAttribute** XmlNode methods.

You don’t have to specifically add a SOAP body section. The body is automatically added when you call the AddMethod method.

The **AddParm** method sets the parameters for the parameter element of the method as a name, value pair. Multiple parameters can be set and are used in the order that this method is called. If an XML Schema is defined, types can be used using the **AddAttribute** XmlNode method. Any number of parameter name-value pairs can be added for a method and are added to the XmlDoc in the order that the AddParm methods are called.

You can add parameters to a method only *after* you add the method. The parameters are added in the order you add them in the PeopleCode program.

See Also

Chapter 30, SOAPDoc Class , AddMethod, page 1600

Chapter 30, SOAPDoc Class , AddParm, page 1601

Chapter 34, XmlDoc Class , AddAttribute, page 1776

SOAPDoc Section Access

Use the following properties to access the different parts of a SOAPDoc object. All of these properties return an XmlNode object.

- BodyNode
- EnvelopeNode
- MethodNode

After you have access to the portion of the SOAPDoc that you want, use the XmlNode properties and methods to manipulate the SOAPDoc.

Use the XmlDoc property to convert a SOAPDoc object to an XmlDoc object, and vice-versa.

The **MethodName** property returns the name of the method in the SOAPDoc object.

Note. If you use the MethodName property, you receive only the method name, as a string, not a reference to an XmlNode object that represents the method.

Use the **GetParmName** and **GetParmValue** methods to get the name and the values in a SOAPDoc object for each output parameter. You must use the index number of the parameter you want with these methods. You can determine the total number of parameters with the **ParmCount** property.

See Also

Chapter 30, SOAPDoc Class , BodyNode, page 1605

Chapter 30, SOAPDoc Class , EnvelopeNode, page 1606

Chapter 30, SOAPDoc Class , MethodName, page 1606

Chapter 30, SOAPDoc Class , MethodNode, page 1607

Chapter 30, SOAPDoc Class , GetParmValue, page 1603

PeopleCode to Create a SOAPDoc

The following is the order in which the Soap methods *must* be called in order to create a valid SOAPDoc.

```
&SOAPDoc = CreateSOAPDoc(); /* required */
&SOAPDoc.AddEnvelope(0); /* optional */
&SOAPDoc.AddBody(); /* optional */
&SOAPDoc.AddMethod("GetQuote", 1); /* required */
&SOAPDoc.AddParm("symbols", "PSFT");
&SOAPDoc.AddParm("format", "l1c1d1t1");
&OK = &SOAPDoc.ValidateSOAPDoc(); /* optional */
```

SOAPDoc Fault Section

In the body of a SOAP XML response document, you can define a fault section. Generally this is used if in an inbound message has been processed and some sort of error occurred. The text of the message gives further information about the error.

Use the **AddFault** method to add a fault code and a fault string to the body of a SOAPDoc. Only one fault section can be set within one SOAP XML document.

If you receivea response message that contains a fault, use the **FaultCode** and **FaultString** methods to return these values.

See Also

Chapter 30, SOAPDoc Class , AddFault, page 1597

Chapter 30, SOAPDoc Class , FaultCode, page 1606

Chapter 30, SOAPDoc Class , FaultString, page 1606

The ValidateSOAPDoc Method

Use the **ValidateSOAPDoc** method to validate the SOAP document for correctness as follows:

- Encoded using XML.
- Has SOAP envelope.
- Has SOAP Body.
- Has a SOAP Method.
- Used SOAP encoding and envelope namespaces.

See Also

Chapter 30, SOAPDoc Class , ValidateSOAPDoc, page 1604

SOAPDoc Objects and Messages

To publish the data of a SOAPDoc, you must first transform the SOAPDoc into an XmlDoc object by using the **XmlDoc** property. After you do this, you can use the XmlDoc object with the **SyncRequestXmlDoc**, **PublishXmlDoc**, or **GetXmlDoc** functions, to publish the data either synchronously or asynchronously.

All XmlDoc messages must be associated with a message definition. An XmlDoc message must be associated with an *unstructured* message, that is, a message definition that's created without any record definitions.

Use the SOAPDoc class if all of the following are true:

- Your third-party source or target node requires SOAP-compliant messages.
- Your third-party source or target node requires synchronous transactions.
- Your message conforms to the SOAP specification.

See Also

Chapter 34, XmlDoc Class , page 1757

PeopleSoft Integration Broker, Sending and Receiving Messages

Data Type of a SOAPDoc Object

SOAPDoc objects are declared as type SOAPDoc. For example,

```
Local SOAPDoc &MyDoc;

Global SOAPDoc &SOAPDoc;
```

Scope of a SOAPDoc Object

A SOAPDoc object can only be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

SOAPDoc Built-In Functions

Chapter 1, PeopleCode Built-in Functions, CreateSOAPDoc , page 109

SOAPDoc Class Methods

In addition to the methods listed here, most of the XmlDoc and XmlNode class methods can be used with a SOAPDoc object. The methods are discussed in alphabetical order.

See Also

Chapter 34, XmlDoc Class , page 1757.

AddBody

Syntax

AddBody()

Description

Use the **AddBody** method to add the body section of the SOAP XML document.

This method is optional when you're creating a SOAP document. Use it only if you must set any additional attributes for the body. Otherwise, the body section is automatically generated in the XML document.

If you're going to add a body section, you must do it *after* you've added the envelope section. If you are going to modify the body section, you must do this *before* you add the method sections of the SOAP XML document by calling the BodyNode property and using XmlNode methods and properties.

Note. Only one body section can be set within one Peoplesoft SOAP XML document.

Parameters

None.

Returns

None.

Example

From the following code:

```
&MyDoc.AddBody();
```

the following XML is created:

```
<SOAP-ENV:Body >

</SOAP-ENV:Body>
```

See Also

Chapter 30, SOAPDoc Class , AddMethod, page 1600, Chapter 30, SOAPDoc Class , AddEnvelope, page 1596, Chapter 34, XmlDoc Class , AddAttribute, page 1776.

AddEnvelope

Syntax

AddEnvelope(*SOAP_Schema*)

Description

Use the **AddEnvelope** method to add the envelope element tags and the default schema. The default encoding styles are also set with this method.

Note. Only one envelope section can be set within one Peoplesoft SOAP XML document.

If you're going to be adding an envelope section, you must do this *before* you add the body section or any methods by calling the EnvelopeNode property and using XmlNode methods and properties.

Parameters

Parameter	Description
SOAP_Schema	Specify whether to use the default SOAP schema or the UDDI schema. You can specify either a numeric value or a constant for this property. The values are:

Numeric Value	Constant Value	Description
0	%SOAP_Schema	Use the SOAP schema. This adds the attributes: xmlns:SOAP-ENV="http:/ /schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-ENC="http:/ /schemas.xmlsoap.org/soap/encoding/
1	%SOAP_UDDI	Use the UDDI schema. This adds the attribute: xmlns:SOAP-ENV="urn:uddi-org:api"
3	%SOAP_NoSchema	Used to enable custom Namespace schemas to be added instead of the default SOAP schemas. You must add the SOA_ENV URI before using any other SOAP methods.

Returns

None.

Example

The following example uses the default SOAP schema. The following PeopleCode program:

```
&MyDoc.AddEnvelope(%SOAP_Schema);
```

Creates the following XML:

```
<?xml version="1.0"?>

<SOAP-ENV:Envelope  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/⇒
```

```
 "xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/>

</SOAP-ENV:Envelope>
```

The following example uses a custom SOAP schema. The following PeopleCode program:

```
&MyDoc.AddEnvelope(%SOAP_NoSchema);
XmlNode &EnvNode = &MyDoc.EnvelopeNode( );
&EnvNode.AddAttribute("xmlns:SOAP-ENV="urn:MyCustomSchema.org");
```

Creates the following XML:

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="urn:MyCustomSchema.org>
</SOAP-ENV:Envelope>
```

See Also

Chapter 30, SOAPDoc Class , AddBody, page 1595.

AddFault

Syntax

AddFault(*Fault_Code*, *Fault_String*)

Description

Use the **AddFault** method to add a fault code, with its corresponding fault string, to a SOAPDoc. Use this method only in a Response SOAP XML Document. The Envelope and Body sections of the XML Document are automatically added if they don't exist.

Parameters

Parameter	Description
Fault_Code	Specify the fault code to add, as a string.
Fault_String	Specify the fault string to add, as a sting.

The following fault codes are supported:

Fault Code	Fault String	Description
100	Version Mismatch	The XML version or SOAP version is not supported by the current implementation.

Fault Code	Fault String	Description
300	Client error	An error occurred on the client while processing the SOAP document.
400	Server error	An error occurred on the server while processing the SOAP document.

Returns

None.

Example

The following example program:

```
&SOAPDoc = CreateSOAPDoc();
SOAPDoc.AddFault("400", "Server Error");


/* ===> The following statement executes this instance: */
&FaultCode = &SOAPDoc.FaultCode;
&FaultString = &SOAPDoc.FaultString;
&OK = &SOAPDoc.ValidateSOAPDoc();
```

Creates the following XML:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" ⇒
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
- <SOAP-ENV:Body>
- <SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:400</faultcode>
  <faultstring>SOAP Server Error</faultstring>
  </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

See Also

Chapter 30, SOAPDoc Class , FaultCode, page 1606, Chapter 30, SOAPDoc Class , FaultString, page 1606.

AddHeader

Syntax

AddHeader()

Description

Use the **AddHeader** method to add the header section of the SOAP XML document.

This method is optional when you're creating a SOAP document. Use it only if you want to include a header in your SOAP XML document.

If you're going to add a header, you must do it *before* you've added any other sections, such as the envelope or the body.

After you've created the header, you can access it using the **GetHeaderNode** method. Once you have a reference to the node, you must use the XmlDoc methods to add specifics to the header section such as encription, as well as any required child nodes.

Note. Only one header section can be set within one Peoplesoft SOAP XML document.

Parameters

None.

Returns

None.

Example

The following test adds a header section into a SOAP XML document where all sections are created explicitly:

```
Local SOAPDocs &Sdoc;
Local XMLNode &Node;

/* ===> Add inputs: */
&Sdoc = CreateSOAPDoc(); /* required */
&Sdoc.AddEnvelope(0); /* optional */
&Sdoc.AddHeader(); /* optional */
&Sdoc.AddBody(); /* optional */
&Sdoc.AddMethod("SOAPXml", 1); /* required */
&Sdoc.AddParm("symbols", "PSFT");

&OK = &Sdoc.ValidateSOAPDoc();

/*Append to TEMP file*/
&Xstring = &Sdoc.GenXmlString();
&myfile = GetFile("cmsDoc.xml", "A");
&myfile.WriteLine(&Xstring);
&myfile.Close();

&METHOD = &Sdoc.MethodName;
&ParmCnt = &Sdoc.ParmCount;
&HdrNode= &Sdoc.HeaderNode;

&HdrName = &HdrNode.NodeName; //returns SOAP-ENV:Header

/* Echo out the Returned Outputs */
out_BI_results("SOAP Header test 1");
out_BI_results(" ");
```

```
out_BI_results("Validated: " | &OK);
out_BI_results("Method: " | &METHOD);
out_BI_results("Parm Count: " | &ParmCount);
out_BI_results("Header : " | &HdrName);

out_BI_results("End-of-Test");
```

See Also

Chapter 30, SOAPDoc Class , GetHeaderNode, page 1602

AddMethod

Syntax

AddMethod(*MethodName*, *IsRequest*)

Description

Use the **AddMethod** method to set the name of the method element being processed. Additional attributes for the added method can be set using the **AddAttribute** and **InsertAttribute** method. Executing AddMethod automatically adds any missing Envelope and Body sections.

Note. Only one method can be set within one Peoplesoft SOAP XML document.

The SOAPDoc method can be used only *after* the envelope and body sections have been added.

Set the parameters for the method using the AddParm method.

Parameters

Parameter	Description
MethodName	Specify the name of the method element being added.
IsRequest	Specify if the message is a request message. This parameter takes a numeric value: • 1 if the message is a request message • 0 if the message is a response message

Returns

None.

Example

The following sample program:

```
&MyDoc.AddMethod("GetPrice", 0);
```

Creates the following XML:

```
<GetPrice>

</GetPriceResponse>
```

The following sample program:

```
&MyDoc.AddMethod("GetPrice", 1);
```

Creates the following XML:

```
<GetPrice>

</GetPrice>
```

See Also

Chapter 30, SOAPDoc Class , AddParm, page 1601, Chapter 30, SOAPDoc Class , MethodName, page 1606, Chapter 30, SOAPDoc Class , MethodNode, page 1607, Chapter 34, XmlDoc Class , AddAttribute, page 1776.

AddParm

Syntax

AddParm(*ParmName, ParmValue*)

Description

Use the **AddParm** method to set the parameters for the parameter element of the current method. To set more than one parameter, use the method more than once. The parameters are set and used in the order that this method is called. Any number of parameter name-value pairs can be set for a method and are added to the XML document in the order that this method is called from the PeopleCode program.

Parameters

Parameter	Description
ParmName	Specify the name of the parameter as a string.
ParmValue	Specify the value of the parameter as a string.

Returns

None.

Example

The following PeopleCode:

```
&MyDoc.AddParm("Item", "Apples");
```

Creates the following XML:

```
<Item >Apples</Item>
```

See Also

Chapter 30, SOAPDoc Class , AddMethod, page 1600.

GetHeaderNode

Syntax

GetHeaderNode()

Description

Use the **GetHeaderNode** method to return a reference to the header node. After you have a reference, you can use the XmlDoc methods to read and make changes to the node.

Parameters

None.

Returns

A reference to an XmlDoc node.

Example

The following test adds a header section, validates the SOAPDoc, then access the header.

```
Local SOAPDocs &Sdoc;
Local XMLNode &Node;

/* ===> Add inputs: */
&Sdoc = CreateSOAPDoc(); /* required */

&Sdoc.AddHeader(); /* optional */
&Sdoc.AddMethod("SOAPXml", 1); /* required */

&OK = &Sdoc.ValidateSOAPDoc();

/*Append to TEMP file*/
&Xstring = &Sdoc.GenXmlString();
&myfile = GetFile("cmsDoc.xml", "A");
&myfile.WriteLine(&Xstring);
&myfile.Close();

&METHOD = &Sdoc.MethodName;
&ParmCnt = &Sdoc.ParmCount;
&HdrNode= &Sdoc.HeaderNode; //returns SOAP-ENV:Header

&HdrName = &HdrNode.NodeName;

/* Echo out the Returned Outputs */
out_BI_results("SOAP Header test 1");
out_BI_results(" ");
out_BI_results("Validated: " | &OK);
out_BI_results("Method: " | &METHOD);
out_BI_results("Parm Count: " | &ParmCount);
out_BI_results("Header : " | &HdrName);
```

```
out_BI_results("End-of-Test");
```

See Also

Chapter 30, SOAPDoc Class , AddHeader, page 1598

GetParmName

Syntax

GetParmName(*Index*)

Description

Use the **GetParmName** method access the parameter names for a method.

Parameters

Parameter	Description
Index	Specify the number of the parameter that you want to access. Values start at 1.

Returns

A string.

Example

```
For &I = 1 to &SOAPDoc.ParmCount

&ParmName = &SOAPDoc.GetParmName(&I);
&ParmValue = &SOAPDoc.GetParmValue(&I);

   /* Do processing */

End-For;
```

See Also

Chapter 30, SOAPDoc Class , GetParmValue, page 1603, Chapter 30, SOAPDoc Class , ParmCount, page 1607.

GetParmValue

Syntax

GetParmValue(*Index*)

Description

Use the **GetParmValue** method to access the parameter values for a method.

Parameters

Parameter	Description
Index	Specify the number of the parameter you want to access. Values start at 1.

Returns

A string.

Example

```
For &I = 1 to &SOAPDoc.ParmCount

&ParmName = &SOAPDoc.GetParmName(&I);
&ParmValue = &SOAPDoc.GetParmValue(&I);

   /* Do processing */

End-For;
```

See Also

Chapter 30, SOAPDoc Class , GetParmName, page 1603, Chapter 30, SOAPDoc Class , ParmCount, page 1607.

ValidateSOAPDoc

Syntax

ValidateSOAPDoc([1])

Description

Use the **ValidateSOAPDoc** method to validate the SOAP document for correctness as follows:

SOAP message is:

- Encoded using XML.
- Has SOAP envelope.
- Has SOAP body.
- Used SOAP encoding and envelope namespaces.

If you want to validate whether your SOAP document uses the SOAP-ENV namespace prefix, use ValidateSOAPDoc without any parameters. If you want to ignore namespace prefixes during SOAP validation, specify 1.

Parameters

Parameter	Description
1	Specify whether to check for namespace prefixes. This parameter is optional.

Returns

This method returns either a number or a constant. The values are:

Numeric Value	Constant Value	Description
0	%SOAP_Valid	The SOAPDoc is valid.
1	%SOAP_NoEnvelope	Missing or invalid envelope section in XML document.
2	%SOAP_InvalidEnvelope	Envelope has improper Namespace used. Should be: SOAP-ENV or SOAP-ENC
3	%SOAP_NoBody	Missing body section in XML document.
4	%SOAP_NoMethod	Missing or invalid method section in XML document.
6	%SOAP_InvalidXml	Invalid XML syntax in SOAPDoc.

Example

```
&Return = &MyDoc.ValidateSOAPDoc();
If &Return <> 0 Then
   /* do error processing */
End-if;
```

See Also

Chapter 30, SOAPDoc Class , FaultCode, page 1606, Chapter 30, SOAPDoc Class , FaultString, page 1606.

SOAPDoc Class Properties

In addition to the properties listed here, most of the XmlDoc and XmlNode class properties can be used with a SOAPDoc object. The properties are discussed in alphabetical order.

See Also

Chapter 34, XmlDoc Class , page 1757.

BodyNode

Description

This property returns a reference to the body section of a SOAPDoc, as an XmlNode.

This property is read-only.

See Also

Chapter 34, XmlDoc Class , page 1757.

EnvelopeNode

Description

This property returns a reference to the envelope section of a SOAPDoc, as an XmlNode.

This property is read-only.

See Also

Chapter 34, XmlDoc Class , page 1757.

FaultCode

Description

Use the **FaultCode** property to return the fault code if the SOAP message has one. Not every message has a fault code. It is returned only when the SOAP method has had an error in processing *and* the SOAP method supports fault codes.

This property is read-only.

Example

```
&Fault = &MyDoc.FaultCode;
```

See Also

Chapter 30, SOAPDoc Class , FaultString, page 1606, Chapter 30, SOAPDoc Class , AddFault, page 1597.

FaultString

Description

Use the **FaultString** property to return the fault string if the reply has one. Not every SOAP message has a fault string. It is only returned when the SOAP method has had an error in processing *and* the SOAP method supports fault strings.

Example

```
&Fault = &MyDoc.FaultString;
```

See Also

Chapter 30, SOAPDoc Class , FaultCode, page 1606, Chapter 30, SOAPDoc Class , AddFault, page 1597.

MethodName

Description

This property returns the name of the method of the SOAPDoc, as a string.

If you want a reference to the method that you can work with, as an XmlNode, use the MethodNode property.

This property is read-only.

MethodNode

Description

This property returns a reference to the method section of a SOAPDoc, as an XmlNode.

If you want only the name of the method returned, use the MethodName property.

This property is read-only.

See Also

Chapter 34, XmlDoc Class , page 1757.

ParmCount

Description

This property returns the total number of parameters for the method section in a SOAP XML document as a number.

This property is read-only.

Example

```
For &I = 1 to &SOAPDoc.ParmCount
 &ParmName = &SOAPDoc.GetParmName(&I);
 &ParmValue = &SOAPDoc.GetParmValue(&I);
   /* do processing */
End-For;
```

XmlDoc

Description

This property transforms a SOAPDoc object to an XmlDoc object. This is necessary only when sending or receiving a response message. You do *not* need to do this conversion to use the XmlDoc or XmlNode methods and properties.

This property is read-write.

Example

The following is an example of sending a SOAP message and receiving the response:

```
Local XmlDoc &request, &response;
Local string &strXml;
Local SOAPDoc &soapReq, &soapRes;

/* create the SOAP XML Document */
&soapReq = CreateSOAPDoc();
&soapReq.AddMethod("GetPrice");
&soapReq.AddParm("Item", "Apples");
```

```
/* convert SOAP to XmlDoc */
&request = &soapReq.XmlDoc;

/* Send the Request */
&response = SyncRequestXmlDoc(&request, Message.QE_SOAP_REQ, Node.UNDERDOG);

/* Get the SOAP response from the XmlDoc response */
&soapRes = CreateSOAPDoc();
&soapRes.XmlDoc = &response;
&OK = &soapRes.ValidateSOAPDoc();
&strXml = &soapRes.GenXmlString();
```

SOAPDoc Class Examples

The following are typical example of how you would use a SOAPDoc object.

Creating a SOAP Document

The following is the minimal PeopleCode you need to create a SOAP XML document:

```
Local SOAPDoc &SOAPDoc;

&SOAPDoc = CreateSOAPDoc();
&SOAPDoc.AddMethod("GetPrice");
&SOAPDoc.AddParm("Item", "Apples");
```

The following is an example of creating a SOAP XML document with a simple method.

```
Local SOAPDoc &SOAPDoc;

&SOAPDoc = CreateSOAPDoc();

&SOAPDoc.AddEnvelope(0);
&SOAPDoc.AddBody();
&SOAPDoc.AddParm("symbols", "PSFT");
&SOAPDoc.AddParm("format", "l1c1d1t1");

&OK = &SOAPDoc.ValidateSOAPDoc();
```

Reading an Existing SOAP Document

The following code loads an SOAP XML document from a URL, then finds the method name and parameters to call that method.

```
Local SOAPDoc &SOAPDoc;
```

```
Local Number &ParmCount;
Local Boolean &Result;

&SOAPDoc = CreateSOAPDoc();
&SOAPDoc.ParseXmlFromURL("C:\ptdvl\840S2\files\inputSOAP.xml");

&ParmCount = &SOAPDoc.ParmCount;
For &I = 1 to &ParmCount

   &ParmName = &SOAPDoc.GetParmName(&I);
   &ParmValue = &SOAPDoc.GetParmValue(&I);

/* Do processing */
End-for;
```

Using SOAP XML Text

The following shows a SOAP XML Document text for the following pseudo-code:

```
Item = "Apples"
Price = GetPrice(Item);
```

Example of SOAP Request Header

Here's the HTTP Header for a SOAP message. This Host URL is the server where the method defined in the SOAP envelope is processed:

```
POST /GetPrice HTTP/1.1
Host: www.farmersmarket.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction:"GetPrice"
```

This is a simple example of a SOAP XML Document, requesting the price of Apples. It uses the default SOAP schemas. The name of the method is GetPrice with the parameter Item which has a value of "Apples".

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.SOAP.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.SOAP.org/soap/encoding/"
>
    <SOAP-ENV:Body>
        <m:GetPrice xmnls:m="SomeURI">
        <Item>Apples</Item>
        </m:GetPrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Example of SOAP HTML Reply Header

Here's the HTTP Header for a SOAP message Reply with a 200 return code that the message was processed OK.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
```

This simple example of a SOAP XML Document reply returns the price of Apples as 34. The SOAP reply always codes the name of the requested method with the **Response** tag attached to it, **GetPriceResponse** in this case.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.SOAP.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.SOAP.org/soap/encoding/"
>
   <SOAP-ENV:Body>
       <m:GetPriceResponse xmlns:m="Some-URI">
          <Price>34</Price>
       </m:GetPriceResponse>
     </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Using SyncRequestXmlDoc Example

The following converts the SOAPDoc object into an XmlDoc object, then uses the XmlDoc object to send the data. The response comes back as an XmlDoc object, which is then converted back into a SOAPDoc object.

```
Local XmlDoc &request, &response;
Local string &strXml;
Local SOAPDoc &soapReq, &soapRes;

&soapReq = CreateSOAPDoc();
&bool = &soapReq.ParseXmlFromURL("C:\\ptdvl\\appserv\\QE840U2\\files\\input⇒
SOAP.xml");

&strXml = &soapReq.GenXmlString();
&MY_FILE = GetFile("c:\temp\syncInput.txt", "w", "a", %FilePath_Absolute);
&MY_FILE.WriteLine("YEAH!");
&MY_FILE.WriteLine(&strXml);
&MY_FILE.Close();

&request = &soapReq.XmlDoc;
&response = SyncRequestXmlDoc(&request, Message.QE_SOAP_REQ, Node.UNDERDOG);

&soapRes = CreateSOAPDoc();
&soapRes.XmlDoc = &response;
&OK = &soapRes.ValidateSOAPDoc();
&strXml = &soapRes.GenXmlString();

&MY_FILE = GetFile("c:\temp\sync.txt", "w", "a", %FilePath_Absolute);
&MY_FILE.WriteLine(&strXml);
```

```
&MY_FILE.Close();
```

SoapDoc Class Outbound Synchronous Example

The following example uses the SyncRequestXmlDoc built-in Function.

Note. Because SyncRequestXmlDoc is an XmlDoc function, you must convert your SoapDoc request message to an XmlDoc object before sending it. Likewise, the response message will be an XmlDoc object, so you must convert it to a SoapDoc object to process it using SOAP methods.

```
Local File &MY_FILE;

Local XmlDoc &request, &response;
Local string &strXml;
Local SoapDocs &soapReq, &soapRes;

&soapReq = CreateSoapDoc();
&bool = &soapReq.ParseXmlFromURL("C:\\ptdvl\\appserv\\QE840U2\\files\\input⇒
Soap.xml");
&strXml = &soapReq.GenXmlString();
&MY_FILE = GetFile("c:\temp\syncInput.txt", "w", "a", %FilePath_Absolute);
&MY_FILE.WriteLine("YEAH!");
&MY_FILE.WriteLine(&strXml);
&MY_FILE.Close();

&request = &soapReq.XmlDoc;
&response = SyncRequestXmlDoc(&request, Message.QE_SOAP_REQ, Node.UNDERDOG);

&soapRes = CreateSoapDoc();
&soapRes.XmlDoc = &response;
&OK = &soapRes.ValidateSoapDoc();
&strXml = &soapRes.GenXmlString();

&MY_FILE = GetFile("c:\temp\sync.txt", "w", "a", %FilePath_Absolute);
&MY_FILE.WriteLine(&strXml);
&MY_FILE.Close();
```

SoapDoc Class Inbound Synchronous Example

The following example uses the GetMessageXmlDoc built-in function.

Note. Because GetMessageXmlDoc is an XmlDoc function, you must receive the inbound request message as an XmlDoc object, then convert it to a SoapDoc object to process it using SOAP methods. Likewise, because the request was sent using SyncRequestXmlDoc, you must convert your SoapDoc response message to an XmlDoc object before returning it.

```
Local XmlDoc &request, &response;
Local string &strXml;
```

```
Local SoapDocs &soapReq, &soapRes;

&soapReq = CreateSoapDoc();

&request = GetMessageXmlDoc();
&soapReq.XmlDoc = &request;
&OK = &soapReq.ValidateSoapDoc();
&parmN = &soapReq.GetParmName(1);
&parmV = &soapReq.GetParmValue(1);

&soapRes = CreateSoapDoc();
&soapRes.AddEnvelope(0);
&soapRes.AddBody();
&soapRes.AddMethod("StockPrice", 1);
&soapRes.AddParm(&parmN, &parmV);
&soapRes.AddParm("Price", "29");
&OK = &soapRes.ValidateSoapDoc();

&response = &soapRes.XmlDoc;
ReturnToServer(&response);
```

CHAPTER 31

SQL Class

This chapter provides an overview of the SQL class and discusses the following topics:

- Using Record class SQL.
- Creating a SQL definition.
- Binding and executing of SQL statements.
- Fetching from a Select.
- Using Array of Any for bind values.
- Understanding different "Styles" of SELECT.
- Reusing a cursor.
- Understanding SQL objects and Application Engine programs.
- Declaring a SQL object.
- Scope of a SQL object.
- SQL class built-in functions.
- SQL class methods.
- SQL class properties.

See Also

Chapter 31, SQL Class , Understanding SQL Objects and PeopleSoft Application Engine Programs , page 1621

Chapter 25, Record Class , page 1441

PeopleCode Developer's Guide, Data Buffer Access

Understanding SQL Class

You can create SQL definitions in PeopleSoft Application Designer. These can be entire SQL programs, or just fragments of SQL statements that you want to re-use. PeopleCode provides the SQL class for accessing these SQL definitions in your PeopleCode program at runtime.

The SQL class provides capability beyond that offered by SQLExec. Unlike SQLExec, which fetches just the first SELECTed row, operations for the SQL class allow iteration over all rows fetched. This can dramatically improve performance if you're doing a million operations and you've set the BulkMode property to True.

A list of input (bind) values, and a list of output variables are supported, as they are in SQLExec. The input and output variables are limited to the same PeopleCode types that can be used with SQLExec, with the addition of a new class called Record.

At runtime, you instantiate a record object from the Record class. A record object is a "one row" instantiation of a record definition.

- When used as an output from an SQL object, each next field in the record is populated with the next column returned.
- When used as an input, several fields in the record are used to replace a single bind marker in the SQL statement. The bind marker for a record describes the type of substitution to be done.

Both records and other PeopleCode types can be mixed in both the output and input.

At runtime, you instantiate a SQL object from the SQL class. The SQL object is loaded by either a constructor for the object, or an explicit **Open** method call. Optionally, a SQL constructor and the Open method support setting the SQL statement through a string parameter. This capability enables the creation and execution of ad-hoc SQL statements.

The SQL class has a **Fetch** method for iterating through the rows fetched by a select. A *cursor* is used to control this connection between the runtime SQL object and the database. The cursor is closed automatically when the object goes out of scope. The cursor can be closed before that by using the **Close** method. The status of the connection is available from the Boolean **IsOpen** property.

The general status of operations is available from function or method return values or properties. Detailed status is not available, as the operations are designed to terminate on unexpected errors or errors that cannot be reasonably recovered from by application level logic.

Considerations for Extra Spaces

On the DB2/OS390 platform, when the **zparm** option is set to decimal equals comma (so comma is used as the database decimal separator), extra blanks are added after commas to ensure that they are not mistaken for decimal separators.

Considerations for Case Sensitivity

When processing a SQL statement, the system automatically casts all fieldnames and possibly record names to uppercase when processing a SQL statement. When processing records from a third party, fields that are lowercase get cast into uppercase, which can create a runtime issue on case sensitive platforms.

To prevent this, use the %NoUppercase meta-SQL statement at the beginning of the SQL statement.

See Also

Chapter 2, Meta-SQL , %NoUppercase , page 545

Using Record Class SQL

There are some SQL type operations that you can do with the Record class, such as INSERT, DELETE and UPDATE. The advantage of using the Record class methods is ease of use, re-use of code, and so on. However, if you're doing many iterations of the same operation (like a million UPDATEs) you should use the SQL object with the BulkMode property set to True.

The SQL object maintains a state (that is, a cursor). Hence, if your database can take advantage of BulkMode, instead of a million operations, the commands are "bulked up" and committed only once. This can improve performance dramatically.

Creating a SQL Definition

You can create SQL definitions in PeopleSoft Application Designer, using the SQL editor. These SQL statements can be entire SQL programs, or just fragments that you want to re-use. After you have created a SQL definition, you can use it to populate a SQL object (using FetchSQL, Open, or GetSQL)

You can also create a SQL statement in PeopleCode (using CreateSQL), save it as a SQL definition (StoreSQL), then access it in PeopleSoft Application Designer.

See Also

Chapter 31, SQL Class , Open, page 1626

PeopleCode Developer's Guide, Introducing the SQL Editor

Chapter 1, PeopleCode Built-in Functions, FetchSQL , page 180

Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109

Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271

Chapter 1, PeopleCode Built-in Functions, StoreSQL , page 450

Binding and Executing of SQL Statements

The processing of an SQL statement involves a series of steps.

1. The binding process is the replacement of (variable) input values in the statement, in places indicated by bind placeholders.

   The input values are substituted into the SQL statement in place of the bind placeholders. These placeholders have the form "*:number*", or "*%bindop*(*:number* [, *parm*]. . .)" where the *number* indicates which input value is to be substituted, and the *bindop* and *parm* strings indicate what meta-SQL binding function is to be performed.

   Note. There must be no spaces between the *bindop* and the left parenthesis.

The following binding meta-SQL functions are used with record objects to substitute various forms of fieldnames and values into the SQL statement. The goal of these binding functions is to enable the writing of SQL and PeopleCode that can manipulate records without dependencies on the exact fields in the records.

- %DTTM
- %InsertValues
- %KeyEqual
- %KeyEqualNoEffDt
- %List
- %OldKeyEqual
- %Table
- %UpdatePairs

2. The execution of an SQL statement is the carrying out of the operation of the statement by the database engine.

   This view of SQL statement processing is actually simpler than what actually occurs. In actual practice, the binding occurs in two distinct phases. The database engine (or its supporting routines) is aware of only the latter phase. For some operations, some database engines are able to delay the second stage of binding and the execution of an SQL statement, so the statement can be rebound and re-executed (with different input values). The advantage of this is that these bindings and executions can be accumulated and transmitted across the network to the database server, with several database operations being done in one network operation. This is sometimes referred to as *bulk mode*. Because the network time consumes most database time operations, the performance advantages of bulk mode can be significant.

   However, in bulk mode, individual operations might not be executed immediately by the database engine. The result is that the application might not see errors that arise until later operations are performed.

   SQL SELECT statements are not bound multiple times, rather the retrieved rows are accumulated and sent across the network many rows at a time, also decreasing the effect of network delays.

See Also

Chapter 2, Meta-SQL , page 511

Fetching From a Select

Some operations fetch a row of data into a list of output variables. The values of the fields from the select row are assigned in order to the given output variables. If one of these is a reference to a PeopleCode record object, the fields in the record are assigned the successive values from the row of the select, until all the fields are assigned. Assignment then continues with the next output variable, if any. The number of output fields and variables must equal the number of fields in the row of the select.

See Also

PeopleCode Developer's Guide, Data Buffer Access

Using Array of Any for Bind Values or Fetch Results

You can use a parameter of type "Array of Any" in place of a list of bind values or in place of a list of fetch result variables for the Execute, Fetch, and Open methods. This is generally used when you don't know how many values are needed until the code runs.

For example, suppose that you had a SQL definition INSERT_TEST, that dynamically (that is, at runtime) generated the following SQL statement:

```
"INSERT INTO PS_TESTREC (TESTF1, TESTF2, TESTF3, TESTF4, . . . N) VALUES (:1, :2,⇒
 %DateTimeIn(:3), %TextIn(:4)), . . . N";
```

Suppose you have placed the values to be inserted into an Array of Any, say &AAny:

```
&AAny = CreateArrayAny("a", 1, %DateTime, "abcdefg", . . . N);
```

You can execute the insert with the following:

```
&AAny = CreateArrayAny("a", 1, %DateTime, "abcdefg", . . . N);
&SQL = GetSQL(SQL.INSERT_TEST);
While /* Still something to do */
   &SQL.Execute(&AAny);
End-While;
&SQL.Close();
```

Because the Array of Any promotes to absorb any remaining select columns, it must be the last parameter for the SQL object Fetch method or (for results) SQLExec. For binding, it must be the only bind parameter, as it is expected to supply all the bind values needed.

Understanding Different "Styles" of SELECT

Using the SQL object, you have different methods of using a SELECT statement in your code, depending on your application.

The most direct way is with an Open and Fetch. This fetches successive rows, and you can process each row after it's fetched.

The pseudo code for this type of select statement is:

```
&SQL.Open("Select .  .  .", bindpars);
While &SQL.Fetch(resultpars)
   /* process result row */
End-While;
```

Another method is to set up the loop based on something you want to change, and while you still want to change things, continue the loop.

The pseudo code for this type of select statement is:

```
&SQL.Open("Update/Insert/Delete/other.  .  .  :n .  .  .");
```

```
/* Bind references, but no bind parameters */
While you_want_to_do_changes
   /* set up for changes */
 &SQL.Execute(bindpars);
End-While;
```

Reusing a Cursor

Reusing a cursor means *compiling* a SQL statement just once, but *executing* it more than once. When a SQL statement is compiled, the database checks the syntax and chooses a query path. By doing this only once, but executing this statement several times, you can obtain an improvement in performance.

In PeopleCode, you can reuse a cursor either by using the ReuseCursor property or by following certain restrictions in your code. In this section, we discuss how to:

- Use the ReuseCursor property.
- Reuse a cursor without the ReuseCursor property.

Using the ReuseCursor Property

If you specify the ReuseCursor property as True, the cursor isn't closed until either the SQL object is explicitly closed or re-opened. This provides greater control over the cursor associated with your SQL object. However, when you set ReuseCursor to True, you're essentially pledging to do the right thing in your program. There are some considerations for how you use this property:

In a PeopleSoft Application Engine program, if your program can be restarted, you *must*check for a restart after a checkpoint by testing to see if the SQL object is open after the checkpoint. If it isn't open, that means a restart has happened, and you must reopen the SQL object. In most cases, on checkpoints, your open SQL objects aren't closed, saving the overhead of re-establishing the SQL object after the checkpoint. If the SQL object is open on a select statement at a checkpoint it isn't restored to the open state because you cannot reliably establish the state of the execute-fetch sequence.

In the following example the SQL object is established on a select statement which is executed twice with different bind parameters but is compiled only once. Without the ReuseCursor property the SQL object would be closed after the first fetch cycle completes.

```
Local SQL &Sql;
Local String &Company1 = "CCB";
Local String &Company2 = "CCF";

&Sql = CreateSQL("SELECT A.COMPANY, %Timeout(A.EFFDT) FROM %Table(COMPANY_TBL) A⇒
 WHERE A.COMPANY=:1", &Company1);
/* commenting this out should make the subsequent Execute fail */
&Sql.ReuseCursor = True;

While &Sql.Fetch(&Company1, &Effdt)
 /* processing for the first Company */
```

```
End-While;

&Sql.Execute(&Company2);

While &Sql.Fetch(&Company1, &Effdt)
/* processing for the second company */

End-While;

&Sql.Close();
```

Reusing a Cursor Without the ReuseCursor Property

If you want to reuse a cursor in your program, there are several conditions:

- You have to use the SQL object. Only SQL objects retain SQL cursor information.
- For INSERT, DELETE, and UPDATE statements, you automatically reuse the cursor as long as you don't change the SQL statement as part of the binding process. If the SQL statement is textually the same, so only the binds have changed, you get reuse. For example, you won't get reuse if you first bind one kind of record object to %Insert(), then bind another, different kind of record object.
- For SELECTs, you *must*use the meta-SQL shortcuts (%SelectAll, %SelectByKey, or %SelectByKeyEffDt), *and* the CreateSQL or GetSQL functions without supplying any bind or buffer parameters. The bind parameters are supplied in the Execute function. The Fetch parameter must be the fetch buffer, and be the same as the first Execute parameter. You can supply WHERE clauses, ORDER-BY, and so on, on the end of the SQL string containing the meta-SQL.

Note. BulkMode doesn't reuse the cursor in the same way as the SQL object. BulkMode requests that, when the system can reuse the cursor, it also holds all the changes so they can be communicated to the database in batches.

By using one of the forms of the Select meta-SQL, you're guaranteeing the resulting fetched values are all put into one record object (buffer). This means the implementation doesn't have to ask the database for the length and type of each column: the record buffer is already defined. So this sample code uses the SQL object to maintain the state of the connection with the database, and the record object, to maintain a series of fields suitable for database operations.

```
Local SQL &SQL;
Local Record &REC;

&REC = CreateRecord(Record.KP_KPI_DFN);

/* start with select statement, no bind refs, no */
/* bind parameters */

&SQL = CreateSQL("%Selectall(:1) Where SETID = :2 and KPI_ID = :3 and EFFDT = ⇒
(SELECT MAX(EFFDT) FROM PS_KP_KPI_DFN WHERE SETID = :4 AND KPI_ID = :5 AND EFFDT <=⇒
 %DateIn(:6))");
```

Start Loop

```
/* bind and execute the statement */

&SQL.Execute(&REC, &SETID_KPI, &COMPID, &SETID_KPI, &COMPID, &EFFDT);

 /* Note record parameter for Fetch statement must be
the same as the first Execute parameter
the results are in this record */

 If &SQL.Fetch(&REC) Then

  /*process this record */

 End-If;
```

End Loop

```
&SQL.Close(); /* there is no implicit close on a Fetch returning False */
```

The following example comes from a PeopleSoft Application Engine program. Because the loop goes in and out of PeopleCode, you must declare the SQL object as Global. This example is in three parts.

1. Program called by Application Engine before the loop:

```
Global SQL &SQL;

&SQL = CreateSQL("INSERT INTO %Table(MY_WORK) (TRANS , REGISTERWRITE) VALUES (:1, ⇒
:2)");

&SQL.BulkMode = True; /* not required for reuse, but will get better performance⇒
 on platforms that support bulk insert */
```

2. Program called in the Application Engine loop on SELECT 'X' FROM PSRECDEFN WHERE RECNAME LIKE '%A':

```
Global SQL &SQL;

&var1 = "X";
&var2 = "Y";

&SQL.Execute(&var1, &var2);
```

3. Program called by Application Engine after the loop:

```
Global SQL &SQL;

&SQL.Close();
```

Understanding SQL Objects and PeopleSoft Application Engine Programs

A global variable won't go out of scope until a PeopleSoft Application Engine program finishes. However, all SQL objects are forced closed (that is, the cursor closed) sooner than that. *Any open SQL object is forced closed just before any checkpoint in an Application Engine program.*This is to ensure that the application can be restarted successfully from the checkpoint. After the SQL object is closed, you can reopen the SQL object, or query its properties, (**Status**, **IsOpen**). The simplest way to avoid unnecessary closing of the SQL object is to set the ReuseCursor property to True. A restartable program should always check that SQL objects are open before using them in steps where it's expected they're open. In the absence of an intervening checkpoint, an open SQL object remains open until the Application Engine program finishes.

Declaring a SQL Object

SQL objects are declared as type SQL. For example:

```
Local SQL &MYSQL;

Global SQL &MySql = CreateSQL(SQL.MySql);
```

Scope of an SQL Object

An SQL object can be instantiated only from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

Your SQL statements sometimes change the database. When your SQL changes the database, your code should be only in one of the following events:

- SavePreChange
- WorkFlow
- SavePostChange
- Message Subscription
- FieldChange
- Application Engine PeopleCode action

SQL Class Built-in Functions

Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109

Chapter 1, PeopleCode Built-in Functions, FetchSQL , page 180

Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271

Chapter 1, PeopleCode Built-in Functions, StoreSQL , page 450

SQL Class Methods

In this section, we discuss the SQL class methods. The methods are discussed in alphabetical order.

Close

Syntax

Close()

Description

The **Close** method closes the SQL object. This terminates any incomplete fetching of select result rows, completes any buffered operations (that is, using **BulkMode**), and disassociates the SQL object from any SQL statement that was open on it.

After **BulkMode** operations, the **RowsAffected** property is not valid.

Parameters

None

Returns

True on successful completion, False if there was a duplicate record error. Any errors associated with buffered operations (that is, using **BulkMode**), other than duplicate record errors, cause termination.

Example

```
&SQL = CreateSQL("%Delete(:1)");
While /* Still something to do */
 /* Set key field values of &ABS_HIST */
 &SQL.Execute(&ABS_HIST);
End-While;
&SQL.Close();
```

See Also

Chapter 31, SQL Class , Open, page 1626, Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109.

Execute

Syntax

Execute(*paramlist*)

Where *paramlist* is an arbitrary-length list of values in the form:
inval1 [, *inval2*] ...

Description

The **Execute** method executes the SQL statement of the SQL object. The SQL object must be open and unbound on a delete, insert, or update statement. That is, the CreateSQL, GetSQL, or Open preceding the Execute must have specified a delete, insert, or update statement with bind placeholders and must not have supplied any input values.

The values in *paramlist* are used to bind the SQL statement before it gets executed.

When using the optional **BulkMode**, the **Execute** operations may be buffered and are not guaranteed to have been presented to the database until a **Close** is done. Thus, in **BulkMode**, errors that arise may not be reported until later operations are done.

Parameters

Parameter	Description
paramlist	Specify input values for the SQL string.

Returns

True on successful completion, False for "record not found" and "duplicate record" errors. Any other errors cause termination.

Example

The following example creates a SQL object for inserting. The statement isn't automatically executed when it's created because there aren't any bind variables. The **Execute** occurs after other processing is finished. The name of the record is passed in as the bind variable in the Execute method.

```
&SQL = CreateSQL("%Insert(:1)");
While /* Still something to do */
   /* Set all the field values of &ABS_HIST.  */
   &SQL.Execute(&ABS_HIST);
End-While;
&SQL.Close();
```

The following example creates two SQL objects, one to be used for fetching, the other for updating the record. The first SQL object selects all the records in the &ABS_HIST record that match &EMPLID. The data is actually retrieved using the Fetch method. After values are set on the record, the update is performed by the **Execute**.

```
&SQL1 = CreateSQL("%Select(:1) where EMPLID = :2", &ABS_HIST, &EMPLID);
&SQL_UP = CreateSQL("%Update(:1)");
While &SQL1.Fetch(&ABS_HIST);
   /* Set some field values of &ABS_HIST.  */
   &SQL_UP.Execute(&ABS_HIST);
End-While;
&SQL_UP.Close();
```

The following is an example of inserting an array of records:

```
Local SQL &SQL;
Local array of Record &RECS;

/* Set up the array of records.  */
.  .  .

  /* Create the SQL object open on an insert */
  /* statement, and unbound */
  &SQL = CreateSQL("%Insert(:1)");
  /* While the array has something in it */
  While &RECS.Len
    /* Insert the first record of the array, */
    /* and remove it from the array.  */
    &SQL.Execute(&RECS.Shift());
End-While;
```

See Also

Chapter 31, SQL Class , Close, page 1622, Chapter 31, SQL Class , BulkMode, page 1627, Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109.

Fetch

Syntax

Fetch(*paramlist*)

Where *paramlist* is an arbitrary-length list of values in the form:
outvar1 [*, outvar2*] ...

Description

The **Fetch** method retrieves the next row of data from the SELECT that is open on the SQL object. Any errors result in termination of the PeopleCode program with an error message.

If there are no more rows to fetch, **Fetch** returns as False, the *outvars* are set to their default PeopleCode values, and the SQL object is automatically closed.

Using **Fetch** with a closed SQL object is processed the same as when there are no more rows to fetch.

Note. If you want to fetch only a single row, the SQLExec function can perform better, as it fetches only a single row from the server.

The return of **Fetch** is not optional, that is, you *must* check for the value of the fetch.

Setting Data Fields to Null

This method will *not* set Component Processor data buffer fields to NULL after a row not found fetching error. However, it does set fields that aren't part of the Component Processor data buffers to NULL. It does set work record fields to NULL.

Parameters

Parameter	Description
paramlist	Specify output variables from the SQL Select statement.

Returns

The result of Fetch is True if a row was fetched. If there are no more rows to fetch, the result is False.

Example

In the following example, the **Fetch** method is used first to process a single row, then to process the ABS_HIST record.

```
Local SQL &SQL;
Local Record &ABS_HIST;

&ABS_HIST = CreateRecord(RECORD.ABSENCE_HIST);
&SQL = GetSQL(SQL.SEL27, 15, "Smith");
While &SQL.Fetch(&NAME1, &BIRTH_DT)
   /* Process NAME1, BIRTHDT from the selected row.  */
End-While;

&SQL.Open(SQL.SEL_ABS_HIST, &NAME1, "Smith");
While &SQL.Fetch(&ABS_HIST)
   /* Process ABS_HIST record.  */
```

The following is an example of reading in an array of record objects:

```
Local SQL &SQL;
Local Record &REC;
Local Array of Record &RECS;

/* Get the SQL object open and ready for fetches.  */
&SQL = CreateSQL("%SelectAll(:1) where EMPLID = :2", RECORD.ABSENCE_HIST, &EMPLID);
/* Create the first record.  */
&REC = CreateRecord(RECORD.ABSENCE_HIST);
/* Create an empty array of records.  *
&RECS = CreateArrayRept(&REC, 0);
While &SQL.Fetch(&REC)
   /* We got a record, add it to the array */
   /* and create another.*/
   &RECS.Push(&REC);
   &REC = CreateRecord(RECORD.ABSENCE_HIST);
End-While;
```

See Also

Chapter 31, SQL Class , Open, page 1626, Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271, Chapter 1, PeopleCode Built-in Functions, SQLExec , page 442.

Open

Syntax

Open(*sql* [, *paramlist*])

Where *paramlist* is an arbitrary-length list of values in the form:
inval1 [, *inval2*] ...

Description

The **Open** method associates the *sql* statement with the SQL object. The *sql* parameter can be either:

- A string value giving the SQL statement.
- A reference to a SQL definition in the form **SQL**.*sqlname*.

If the SQL object was already open, it is first closed. This terminates any incomplete fetching of select result rows, completes any buffered operations (that is, using **BulkMode**), and disassociates the SQL object from any SQL statement that was open on it.

Opening and Processing sql

If *sql* is a SELECT statement, it is immediately bound with the *inval* input values and executed. The SQL object should subsequently be the subject of a series of **Fetch** method calls to retrieve the selected rows. If you want to fetch only a single row, use the SQLExec function instead. If you want to fetch a single row into a PeopleCode record object, use the record **Select** method.

If *sql* is not a SELECT statement, and *either*: there are some *inval* parameters, *or* there are no bind placeholders in the SQL statement, the statement is immediately bound and executed. This means that there is nothing further to be done with the SQL statement and the **IsOpen** property of the returned SQL object will be False. In this case, using the SQLExec function would generally be more effective. If you want to delete, insert, or update a record object, use the record **Delete**, **Insert**, or **Update** methods with the record object.

If *sql* is not a SELECT statement, there are no *inval* parameters, *and* there are bind placeholders in the SQL statement, the statement is neither bound nor executed. The resulting SQL object should subsequently be the subject of a series of **Execute** method calls to affect the desired rows.

Setting Data Fields to Null

This method will *not* set Component Processor data buffer fields to NULL after a row not found fetching error. However, it does set fields that aren't part of the Component Processor data buffers to NULL.

Parameters

Parameter	Description
sql	Specify *either* a SQL string *or* a reference to a SQL definition in the form **SQL.** *sqlname*.
paramlist	Specify input values for the SQL string.

Returns

None.

Example

Generally, you use the Open method only after you've already gotten a reference to another SQL object. SELECT and SEL_ABS_HIST are the names of the SQL definitions created in PeopleSoft Application Designer.

```
Local SQL &SQL;

&SQL = GetSQL(SQL.SELECT);

/* do other processing */

/* get next SQL statement for additional processing */
/* The open automatically closes the previous */
/* SQL statement */

&SQL.Open(SQL.SEL_ABS_HIST, &NAME1, "Smith");
While &SQL.Fetch(&ABS_HIST)
   /* Process ABS_HIST record.  */
End-While;
```

See Also

Chapter 1, PeopleCode Built-in Functions, GetSQL , page 271, Chapter 1, PeopleCode Built-in Functions, CreateSQL , page 109.

SQL Class Properties

In this section, we discuss the SQL class properties. The properties are discussed in alphabetical order.

BulkMode

Description

This property controls the use of bulk mode. Setting this property to True enables the use of bulk mode, and hence removes any guarantee of the synchronous presentation of error status.

Bulk mode is used only with those database connections and operations that support it. Bulk mode can be used with any SQL operation, that is, with INSERTs, DELETEs, or UPDATEs.

If you're using a PeopleSoft Application Engine program, and have set this property to True, the rows inserted in BulkMode are committed at the next database commit in your program.

After **BulkMode** operations, the RowsAffected property is not valid.

The default value for **BulkMode** is False.

This property is read-write.

Example

The following code is an example of inserting an array of records using bulk mode:

```
Local SQL &SQL;
Local array of Record &RECS;

/* Set up the array of records.  */
  .  .  .
  /* Create the SQL object open on an insert */
  /* statement, and unbound.*/
  &SQL = CreateSQL("%Insert(:1)");
  /* Try for bulk mode.  */
  &SQL.BulkMode = True;
  /* While the array has something in it… */
  While &RECS.Len
    /* Insert the first record of the array, */
    /* and remove it from the array.  */
    If not &SQL.Execute(&RECS.Shift) then
      /* A duplicate record found, possibly */
      /* in bulk mode.  There is no way to  */
      /* tell which record had the problem. */
      /* One approach to recovery is to fail*/
      /* the transaction and retry it with a*/
      /* process that does only one record  */
      /* at a time, that is, doesn't use    */
      /* bulk mode.            */
      .  .  .;
    End-If;
End-While;
```

See Also

Chapter 31, SQL Class , Binding and Executing of SQL Statements , page 1615, Chapter 31, SQL Class , RowsAffected, page 1630.

IsOpen

Description

This property returns as True if the SQL object is open on some SQL statement.

This property is read-only.

Example

You might use the following in a PeopleSoft Application Engine program, after a checkpoint. MYSELECT is the name of a SQL definition created in PeopleSoft Application Designer:

```
If Not &MYSQL.IsOpen Then
   &MYSQL.Open(SQL.MYSELECT);
End-if;
```

LTrim

Description

This property specifies whether values read by the Fetch method are trimmed of blanks on the left, except in the following cases:

- For long columns.
- The record is directly used to buffer the incoming data. In the following example, the values aren't LTrimmed - regardless of the setting of the LTrim property.

```
Local Record &Rec = CreateRecord(Record.QA_TEST);
Local SQL &Sql = CreateSQL("%SelectAll(:1)");
&Sql.LTrim = True; /* the default */
&Sql.Execute(&Rec);
While &Sql.Fetch(&Rec)
   /* do processing */
End-While;
```

This property takes a Boolean value. The default value is True. If this property is set to False, the selected values are not trimmed of blanks on the left.

This property is read-write.

Note. The removal of blanks from the right end of fetched values (RTrimming) still occurs for non-long columns.

Example

```
Local Record &Rec = CreateRecord(Record.QA_TEST);
Local SQL &Sql = CreateSQL(%SelectAll(:1)", &Rec);
&Sql.LTrim = False;
While &Sql.Fetch(&Rec)
   /* do processing */
End-While;
```

ReuseCursor

Description

This property specifies whether the SQL object tries to reuse the open cursor. This property takes a Boolean value, True, to reuse the SQL cursor.

If specified as True, the SQL object won't be closed at checkpoints, and any non-SELECT SQL is restored. In addition the SQL object is not closed after a fetch cycle completes. It remains open ready to be executed, perhaps with different bind parameters.

If you use this property in your application program, you *must* close the SQL object explicitly or it is closed when the object goes out of scope (that is, when the program finishes.)

You must instantiate a SQL object first before you can reuse it.

This property is read-write.

See Also

Chapter 31, SQL Class , Reusing a Cursor , page 1618.

RowsAffected

Description

This property returns the number of rows affected by the last INSERT, UPDATE, or DELETE statement of the SQL object. After BulkMode operations, the **RowsAffected** property is not valid.

This property is read-only.

Example

The following code is an example that determines if a delete statement actually deleted anything:

```
Local SQL &SQL;

/* Create the SQL object and do the deletion.  */
   &SQL = CreateSQL("Delete from %Table(:1) where EMPLID = :1",⇒
 RECORD.ABSENCE_HIST, &EMPLID);
   If &SQL.RowsAffected = 0 Then
   /* We didn't delete any rows.  */
   .  .  .
End-If;
```

See Also

Chapter 31, SQL Class , BulkMode, page 1627.

Status

Description

This property returns the status of the last statement executed. You can use either the constant or the numeric value for this property. The values for this property are:

Numeric Value	Constant Value	Description
0	%SQLStatus_OK	No Errors.
1	%SQLStatus_NotFound	Record not found.
2	%SQLStatus_Duplicate	Duplicate Record Found This property is read-only.

Example

The following example determines what went wrong after an update:

```
Local SQL &SQL;
Local Record &NEWREC, &OLDREC;
```

```
/* Create and initialize &OLDREC with the keys of the
record to be updated.  Create and initialize &NEWREC
with the new field values for the record.  */
...;

/* Create and execute the update.  */
&SQL = CreateSQL("%Update(:1, :2)", &NEWREC, &OLDREC);
Evaluate &SQL.Status
When = %SQLStatus_OK
   /* It worked.  */
When = %SQLStatus_NotFound
   /* The OLDREC keys were not found.  */
When = %SQLStatus_Duplicate
   /* The NEWREC keys were already there.  */
End-Evaluate;
```

TraceName

Description

This property enables you to assign a name to a SQL statement that has been created in PeopleCode using CreateSQL. This name is used in the PeopleSoft Application Engine timings trace. This property takes a string value.

Note. You *cannot* associate the TraceName property with the execution of a simple SELECT statement created with CreateSQL. This is because the SELECT is executed when the SQL is created, before it has the TraceName assigned. To do this, create a SQL object instead.

If this property isn't set, it defaults to a substring of the SQL statement, indicating the operation and table, (for example, SELECT PS_VOUCHER_LINE.) It may be useful to set TraceName to indicate the origin of the SQL statement.

This property is read-write.

Example

```
&REC = CreateRecord(Record.VOUCHER_LINE);
&SQL = CreateSQL("%selectall(:1) WHERE BUSINESS_UNIT =:2 AND VOUCHER_ID =:3 AND⇒
 VOUCHER_LINE_NUM = :4");
&SQL.TraceName = "AEPROG.SECT1.STEP1.SQL2";
&SQL.Execute(&REC, MATCHING_AET.BUSINESS_UNIT, MATCHING_AET.VOUCHER_ID, &count);
If &SQL.Fetch(&REC) Then
      &count2 = &count2 + 1;
End-If;
```

The previous example would produce the following in the timings trace.

```
SQL Statement               Count   Time     Count   Time     Count   Time     Time
--------------------------- ------- -------- ------- -------- ------- -------- --------

PeopleCode

AEPROG1.SECT1.STEP1.SQL1                       169    5.371     169    3.083    4.000
AEPROG1.SECT1.STEP1.SQL2                       100    5.371     100    3.083    4.454
                                                                              --------
                                                                                8.454
```

Trace Timings Example

You can use this parameter with the Open statement also. The following is an example of how this works:

```
&Sql1 = CreateObject("SQL");
&Sql1.TraceName = "sql1";
&Sql1.Open("Select %FirstRows(1) 'x' FROM psstatus");
&Sql1.Fetch(&Temp);

&Sql2 = CreateObject("SQL");
&Sql2.TraceName = "sql2";
&Sql2.Open("Select 'x' FROM psstatus");
&Sql2.Fetch(&Temp);
```

Value

Description

This property returns the SQL statement associated with the SQL object as a string.

This property is read-only.

Example

To report an error in a SQL definition, including the actual SQL executed, use the Value property to get the text of the SQL statement:

```
Local SQL &SQL;

/* Execute some SQL.  */
&SQL = CreateSQL(SQL.SOMESQL, &EMPLID);
If &SQL.Status = %SQLStatus.NotFound Then
   /* Get the SQL string used.  */
   &SQLSTR = &SQL.Value;
   /* Report the error.  */
   .  .  .;
End-If;
```

CHAPTER 32

SyncServer Class

This chapter provides an overview of the synchronization server and the SyncServer classes and discusses the following topics:

- Understanding the synchronization server events and SyncServer classes
- Using synchronization server events
- Using attachment events
- Scope of a SyncServer object
- SyncServer reference
- Advanced considerations for synchronization

Understanding Synchronization Server Events and the SyncServer Classes

PeopleSoft Mobile Agent extends the functionality of PeopleSoft Internet Architecture (PIA) to disconnected mobile devices, allowing users to continue working with their PeopleSoft applications on a laptop computer or personal digital assistant (PDA) while disconnected from the internet or local network.

The mobile events control the PeopleCode programs used with the mobile device. This PeopleCode uses the mobile classes to control the changes an end user makes to the mobile data.

See *PeopleTools Mobile Agent PeopleBook*.

The synchronization server events control the synchronization of mobile data with the PeopleSoft database. The SyncServer class methods and properties are used for data selection and validation during synchronization.

Understanding the Synchronization Events

When the mobile device is synchronized with the PeopleSoft database, the following events occur on the synchronization server. All of these events only fire when any PeopleCode is associated with the event.

Event Name	Description
OnConflict	This event only occurs when an object has been updated in both the PeopleSoft database and on the mobile device, and there is PeopleCode associated with the event. This event is used for additional refinement of the conflict algorithm.
OnGetDefinition	This event is used for defining Component Interface objects accessed only by a PeopleCode reference.
OnSelect	This event is used for specifying what data should be downloaded (distributed) to the mobile device.
OnValidate	Occurs when any PeopleCode is associated with the event *and* no PeopleCode is associated with the OnValidateSet event. This event is only used for data validation of a medium to large number of objects.
OnValidateSet	This event is used for data validation of a very large number of objects.

In addition, if you create references in the synchronizable Component Interface to be used with attachments, the following events are also available:

Event Name	Description
OnGetProperty	This event is used to specify how attachments are accessed and sent to the mobile device.
OnGetPropertyFilter	Occurs before the OnGetProperty event. This event is used for specifying filter rules to an individual attachment.
OnSetProperty	Occurs when the value of an attachment has changed on the mobile device. This event can is to specify which attachments are synchronized from the mobile device to the PeopleSoft database.

See Also

Chapter 32, SyncServer Class, Using the Synchronization Server Events , page 1636

Understanding the SyncServer Class

During synchronization, use the **%SyncServer** system variable to instantiate and populate a SyncServer object. You can then use this object in one of the synchronization events to specify the details of the current synchronization.

The %SyncServer variable is ignored outside of the synchronization server events, or events called by the synchronization server.

PeopleSoft recommends using the **Synchronizing** property in events outside of the synchronization server events to separate code that should only be executed during synchronization. For example, suppose you had additional business logic to apply to data coming from a mobile device. You could put the following in one of the Component Interface events that gets called during synchronization, such as SavePreChange.

```
If %SyncServer.Synchronizing Then
  /* more data rules here */
End-If;
```

Considerations Using Synchronization Events and the SyncServer Class

The synchronization events are not required for synchronizing a mobile device. You should only start using the SyncServer class methods and properties in the events after your mobile application is running. The following is the typical development cycle for synchronization:

1. Create and test your component.

   If you are using an existing component, you must still test it to verify that it runs without errors and returns the data you are expecting.

2. Create and test your Component Interface.

   If you are using an existing Component Interface, you must still test it to verify that it runs without errors and returns the data you are expecting.

3. Create and test your mobile pages.

   Once your Component Interface is running, *then and only then* should you make the Component Interface synchronizable and build mobile pages based on it.

   Note. Remember that any changes made to the underlying Component Interface are *not* automatically reflected in any mobile pages built from that Component Interface.

   At this point, you should verify whether the search page associated with the component returns the data you need for your mobile application. If it does, and if you are not dealing with a large number of instances of the Component Interface that need to be synchronized, you are finished. There is no need to go any further and create PeopleCode for synchronization.

4. Refine the conflict resolution for your mobile application.

   The properties for the synchronizable Component Interface specify which version of the data should be used if there is a conflict (device always wins, database always wins, and so on.) Use the OnConflict event to refine the conflict rules if necessary.

5. Refine the data used with your mobile application.

If the search record associated with the component is not adequate for returning the data you need for your mobile application, create PeopleCode associated with the OnSelect event to better define the data for your mobile device.

Note. Remember, this event fires when *any* PeopleCode is associated with the event, even if the PeopleCode program is commented out.

6. Fine tune your mobile application.

   If your mobile application requires a medium to large number of objects, between 100–1000 instances of a Component Interface, you can use the OnValidate event to possibly enhance synchronization performance.

   If your mobile applications requires a large number of objects, over 1000 instances of a Component Interface, you can use the OnValidateSet event to possibly enhance synchronization performance.

   If you need to create references to Component Interface objects in PeopleCode, without defining the relationship in Application Designer, you can use the OnGetDefinition event.

Using the Synchronization Server Events

Paleobiologist Mobile includes synchronization server events to support mobile device synchronization and data distribution. If a component interface is synchronizable, these events exist in its associated PeopleCode.

This section discusses how to use:

- OnConflict
- OnSelect
- OnValidate

See Also

PeopleSoft Optimization Framework Guide, Creating the Optimization PeopleCode, OptEngine Objects

Chapter 32, SyncServer Class, Advanced Considerations for Synchronization, page 1649

Using OnConflict

The OnConflict event is invoked during synchronization when an object has been updated independently in the two replicas (server and mobile device; synchronizable Component Interface and mobile) being synchronized—that is, when the two replicas have properties that have been modified.

Note. If a PeopleCode program exists in OnSelect but is commented out, the PeopleCode is not invoked and nothing is pushed into the SyncIDs array. However, because a PeopleCode program exists, the synchronization server assumes that the SyncIDs array is correctly populated with no SyncID values. This may cause unexpected results in your application (all data removed from mobile device, then replaced, and so on.)

In the properties for the synchronizable Component Interface, you select the conflict resolution algorithm that specifies what should happen when any conflict happens, such as device wins, server wins, and so on.

In OnConflict PeopleCode, if there is a conflict, you can select a different conflict resolution algorithm or further refine the reconciliation of the two replicas by determining a new state for the object that appropriately reflects some combination of the two versions.

The OnConflict event uses the SyncServer class properties **ConflictStatus** and **ConflictAlgorithm**. ConflictStatus is generally used to change the conflict resolution algorithm for a single property. ConflictAlgorithm is used to change the conflict resolution algorithm for a device. It is generally used at level 0 of a Component Interface.

OnConflict PeopleCode Example: CONFLICTALGORITHM

In this example, if the mobile device is a personal digital assistant (PDA), this code causes changes on the mobile device to always be accepted.

```
If (%SyncServer.ClientPlatform = %MobileDevice_PDA) Then
  /* If mobile device is a PDA, always accept changes from the  device during⇒
 synchronization */
  %SyncServer.ConflictAlgorithm = %CONFLICTALGORITHM_DEVICEWINS;
End-If;
```

In this example, the program causes the property values from the mobile device owning the property to take precedence.

```
Function isOwner() Returns boolean
  Local boolean &bOwner;
  /* Use user profile information to determine if the
  sync user is the owner of this synchronized Component Interface object... */
  Return &bOwner;
End-Function;

If (isOwner()) Then
  /* No conflict, always accept changes from the owner's device during⇒
 synchronization... */
  %SyncServer.ConflictAlgorithm = %CONFLICTALGORITHM_DEVICEWINS;
Else
  /* Reject changes from non-owners when a conflicts occur...  */
  %SyncServer.ConflictAlgorithm = %CONFLICTALGORITHM_SERVERWINS;
End-If;
```

See Also

Chapter 32, SyncServer Class, ConflictAlgorithm, page 1644

Using OnSelect

The OnSelect event is invoked during synchronization to implement custom data distribution.

To determine which data to send to the mobile device during synchronization, the synchronization server searches for the results of the OnSelect event as it is defined for each Component Interface. In the Component Interface, you can develop PeopleCode in the OnSelect event to select the record rows based on filtering, such as the mobile device owner, person logged in, sales region, and so on.

The OnConflict event uses the SyncServer class property **SyncIDs**, which is an array of synchronization Ids (from the SYNCID field.)

Using a SQL statement, select the SYNCID field for each row or instance of data to be synchronized, and pass this array of SYNCIDs in the **SyncIDs**property array to the synchronization server for retrieval. The OnSelect event runs only if the Component Interface mobile page is a registered portal registry, that is, the Component Interface mobile page must be accessible through a menu option on the mobile device.

If the OnSelect event definition contains no PeopleCode, the default search record for the component interface is used to return an unfiltered array of Sync Ids. All data that satisfies the search record for the component interface is synchronized to the mobile device.

If a PeopleCode program exists in OnSelect but is commented out, the PeopleCode is not invoked and nothing is pushed into the SyncIDs array. However, because a PeopleCode program exists, the synchronization server assumes that the SyncIDs array is correctly populated with no SyncID values.

If PeopleCode exists within the OnSelect event definition, the PeopleCode is run and the resulting set of synchronization Ids is used to determine which data is returned to the mobile device. You can define PeopleCode functions containing data distribution rules for reuse or write code that is specific to each Component Interface.

Using the Common Component Function DistributeDataByRules

This function, contained in the FuncLib FUNCLIB_ECMOBIL, can be used in the OnSelect event to filter data to the mobile device.

OnSelect PeopleCode Examples

In this example, the following PeopleCode on an OnSelect event populates the SyncIDs array with all of the sync Ids from the record rows that have an active lead status (LEAD_STATUS), which is a component interface property. This could be used to supply each salesperson with only the leads that are assigned to that salesperson.

```
/* Select Active Leads */
&sql = CreateSQL("SELECT SYNCID FROM PS_RSF_LEAD WHERE LEAD_STATUS = :1", 'A');
While &sql.Fetch(&SyncID)
  %SyncServer.SyncIDs.Push(&SyncID);
End-While;
```

In the following example, the system variable **%UserId** associates an end user with the PERSON_ID field in the record row. The PeopleCode for the OnSelect event selects all of the synchronization Ids (and associated record rows) that are owned by that user.

```
/* Determine User's Person ID */
SQLExec("SELECT PERSON_ID FROM PSOPRALIAS WHERE OPRID = :1", %UserId, &PersonID);
/* Select Leads with the User's Person ID */
&sql = CreateSQL("SELECT SYNCID FROM PS_RSF_LEAD WHERE PERSON_ID = :1", &PersonID);
While &sql.Fetch(&SyncID)
  %SyncServer.SyncIDs.Push(&SyncID);
End-While;
```

OnSelect and Spidering

Through spidering, rows of data can be required for the mobile device in addition to those selected by the OnSelect event. Suppose that:

- A contact list on a mobile device includes three people: two from the USA states of California and Illinois and one from the Canadian province of Quebec.
- The OnSelect event for state calls for only states or provinces that are in the USA.

The synchronization server retrieves all of the USA states, including California and Illinois, but it also retrieves Quebec because Quebec is already present on the mobile device.

See Also

Chapter 32, SyncServer Class, SyncIDs, page 1648

Using OnValidate

The OnValidate event is invoked during out-of-date detection during full synchronization. Use it only when you have medium to large set of data, approximately 100–1000 instances of a Component Interface.

Generally, when a Component Interface instance is accessed for synchronization, the Compoent Interface keys are set and the **Get** method is called on the Component Interface. If you use OnValidate (or OnValidateSet), this call to the Get method is bypassed. This means that the normal initializtion events associated with a Compoent Interface (PreBuild, RowInit, and so are) are also bypassed. This can improve performance. This implies that you may want to use OnValidate (or OnValidateSet) if the performance cost of doing the Get (and running the other events) is greater than running SQL to compute the current version of the instance.

Note. OnValidate is not invoked during upload changes and upload application, only during full synchronization.

Use the OnValidate event to compute the current version for the synchronized Component Interface object using the SyncServer **validateID** property, and return the result to the synchronization server using the SyncServer **validateVersion** property.

If the Component Interface instance does not exist or is not relevant for some other reason (for example, security), not setting a value into version is interpreted as a validate status of No Relevance and the mobile object is removed from the mobile device.

Use the OnValidate event to get the current version for the object and either set that object to be relevant or remove it from the mobile device.

The OnValidate event uses the SyncServer properties **validateID** and **validateVersion**.

Considerations Using OnValidate

If you create PeopleCode for this event, you must maintain it. Anytime you expose an additional collection in the Component Interface, or remove a collection, you must change your OnValidate PeopleCode.

If you have any PeopleCode in the OnValidateSet event, even if it is commented out, the OnValidate event does not run.

OnValidate runs on every instance of the Component Interface. This means that any code in this event fires once per Component Instance instance.

OnValidate PeopleCode Example

The following PeopleCode sets validation for this instance of the QE_DEMO_CONTACT component interface.

```
SQLExec("SELECT LASTUPDDTTM FROM %Table(QE_DEMO_CONTACT) WHERE SYNCID=:1",
```

```
%SyncServer.validateID, %SyncServer.validateVersion);
```

See Also

Chapter 32, SyncServer Class, validateID, page 1648

Chapter 32, SyncServer Class, validateVersion, page 1649

Using Attachment Events

PeopleTools supports attachments for mobile. The attachment consists of header information (such as size, type, description) and the corresponding document object. The attachment events occur when the mobile user synchronizes the mobile device.

Mobile supports attachments that can be source to and from many different technologies. For example, a relational database, FTP server, or document management system are all possible sources of attachment data.

Do the following to use attachments:

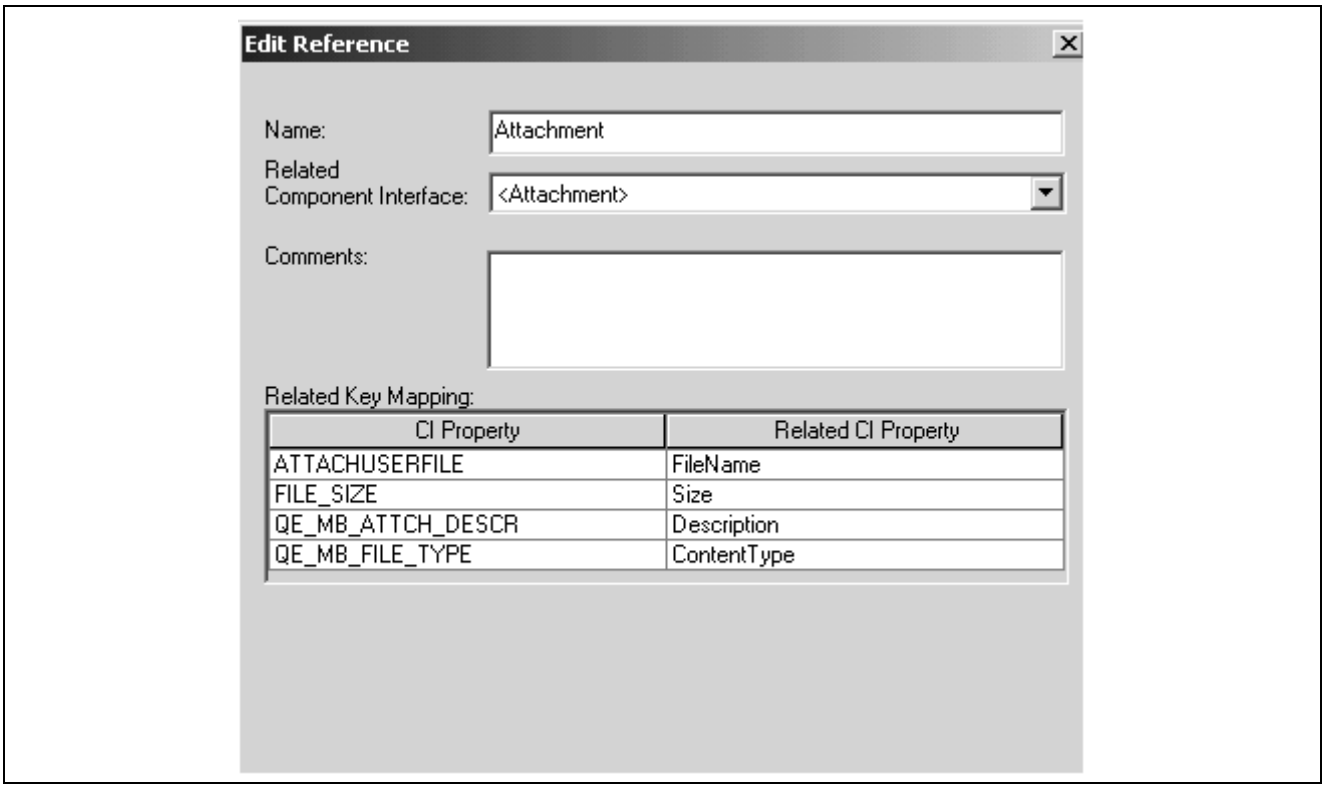
1. Build a synchronizable Component Interface on a component that is setup to use attachments.

   See PeopleCode Developer's Guide, Using Methods and Built-in Functions, Using the Attachment Functions.

2. Create a reference property in the synchronizable Component Interface.
3. Select the reference property, right-click, and select Edit Reference from the popup menu.
4. In the Edit Reference property page, specify the related component interface for the ATTACHMENT reference property as <Attachment>.
5. In the Edit Reference property page, associate the FileName with the ATTACHUSERFILE field.

Once you have created the ATTACHMENT property, all the attachment events are displayed in the PeopleCode editor for the ATTACHMENT property.

See *PeopleTools Mobile Agent PeopleBook*, Using Synchronizable Component Interfaces.



Attachment Edit Reference property page

Both binary and text document types are supported. Developers can define their own storage interaction mechanisms. For example, they can directly retrieve a Calico project file from Calico's native storage mechanism, or move content to and from a document management system such as Documentum.

Note. It is the responsibility of the developer to add filtering programs in case the user tries to load malicious programs onto the system (not allowing .exe files, files that may contain macros or HTML scripts, and so on.)

OnGetProperty Event

The synchronization server invokes the OnGetProperty event when the value for the attachment property is requested. The specific implementation of this event is defined by the developer, and based on the application requirements. It can be used to specify which attachments are downloaded to the mobile device.

The following is a sample of PeopleCode that could be used in the OnGetProperty event. In the example, the **PropertyValue** property is used in the GetAttachment function to specify the filename and location of the attachment file on the synchronization server.

```
If QE_MB_CUST_NEED.FTPSITE <> "" And
      QE_MB_CUST_NEED.ATTACHSYSFILENAME <> "" Then
   %SyncServer.PropertyValue = "c:\PSMobile_FileAttach\" |
       QE_MB_CUST_NEED.ATTACHSYSFILENAME;
   &Result = GetAttachment(QE_MB_CUST_NEED.FTPSITE, ⇒
QE_MB_CUST_NEED.ATTACHSYSFILENAME,
      %SyncServer.PropertyValue);
End-If;
```

OnGetPropertyFilter Event

The synchronization server invokes the OnGetPropertyFilter event prior to invoking the OnGetProperty event to allow the application to apply filtering rules to an individual attachment.

If the SyncServer **ExcludeProperty** property is set to True, the synchronization server only sends the header of the attachment to the mobile device. It does not send the contents of the attachment.

Here is a sample of PeopleCode that could be used in the OnGetPropertyFilter event.

```
If ((%SyncServer.ClientPlatform = %MobileDevice_PDA) And
   (SERVICEORDER_DAMAGEDPART.SIZE > &nMaxAttachmentSize)) Then
   %SyncServer.ExcludeProperty = True;
End-If;
```

OnSetProperty Event

The synchronization server invokes the OnSetProperty event when the value for the attachment is modified. The specific implementation of this event is based on application requirements. It can be used to specify whether the attachment is uploaded from the mobile device.

Here is a sample of PeopleCode that could be used in the OnSetProperty event. The PutAttachment function returns the location of the attachment file in PropertyValue.

```
If QE_MB_CUST_NEED.FTPSITE <> "" And
      QE_MB_CUST_NEED.ATTACHSYSFILENAME <> "" Then
     &Result = PutAttachment(QE_MB_CUST_NEED.FTPSITE, ⇒
QE_MB_CUST_NEED.ATTACHSYSFILENAME,
      %SyncServer.PropertyValue);
End-If;
```

Scope of SyncServer Objects

A SyncServer object can only be instantiated as Local. SyncServer objects are declared using the SyncServer data type. For example:

```
Local SyncServer &MySyncServer;
```

You can also declare and set the SyncServer variable in the same line. For example:

```
Local SyncServer &MySyncServer = %SyncServer;
```

SyncServer Class

The SyncServer object is accessed using the **%SyncServer** system variable.

SyncServer Class Methods

This section describes the SyncServer class methods.

AddReference

Syntax

AddReference(*SyncID*[, *CIDefnName*])

Description

Use the **AddReference** method to identify every instance of an ad-hoc referenced Component Interface that you identified using the AddReferenceType method. Use AddReference in one of the Component build events (PreBuild, PostBuild, RowInit).

Parameters

Parameter	Description
SyncID	This is a string containing the synchronization ID (SyncID).
CIDefnName	This is a string containing the name of the referenced Component Interface. Preface this string with the reserved word **CompIntfc**.

Returns

None.

Example

```
%SyncServer.AddReference(&SyncID, CompIntfc.RefCI);
```

See Also

Chapter 32, SyncServer Class, Advanced Considerations for Synchronization, page 1649

AddReferenceType

Syntax

AddReferenceType(*CIDefnName*)

Description

Use the **AddReferenceType** method in the OnGetDefinition event to identify every ad-hoc reference Component Interfaces. You must have one call for every instance of an an-hoc Component Interface type in the current Component Interface.

Parameters

Parameter	Description
CIDefnName	This is a string containing the name of the referenced Component Interface. Prefix this string with the reserved word **CompIntfc**.

Returns

None.

Example

```
%SyncServer.AddReferencedType(CompIntfc.RefCI);
```

SyncServer Class Properties

This section describes the SyncServer class properties.

ClientPlatform

Description

This property specifies the type of mobile device.

Value	Description
%MobileDevice_PDA	A PDA.
%MobileDevice_Laptop	A laptop computer.

This property is read-write.

ConflictAlgorithm

Description

This property can be used to specify the algorithm for resolving synchronization conflicts. This property is valid only with the OnConflict event. This property is generally used with the level 0 of a Component Interface.

Value	Description
%CONFLICTALGORITHM_ SERVERWINS	The server always wins.
%CONFLICTALGORITHM_ DEVICEWINS	The mobile device always wins.
%CONFLICTALGORITHM_ MERGE	The server and mobile device values are combined where property-level conflicts have not occurred.

Value	Description
%CONFLICTALGORITHM_CUSTOM	No automatic conflict resolution is attempted. Write conflict resolution PeopleCode for the OnConflict event.

This property is read-write.

Merging Values With %CONFLICTALGORITHM_MERGE

All values from the synchronization request are placed into the synchronized Component Interface object on the server, discarding the existing values on the server. Fields that are not part of the data downloaded to the mobile device, but have changed on the server, are retained.

For example, in the following table, Department is modified on the mobile device, Region and Sales Rep (sales representative) are modified on the server, and Sales Rep is not one of the values that is downloaded to the mobile device. The resulting variant has the mobile device values for Department and Region and the server value for Sales Rep. Although a conflict is detected, the mobile device values are used. No message is issued to the mobile device.

Field	Mobile Device Values	Server Values	Results
Department	2222*	1111	2222*
Customer	99999	99999	99999
Region	West	East*	West
Sales Rep	Not on device	444*	444*

* An asterisk indicates values that have changed since the last sync.

ConflictStatus

Description

This property sets the conflict status. This property is valid only with the OnConflict event.

Value	Description
%CONFLICTSTATUS_NOCONFLICT	Remove from the list of conflict.
%CONFLICTSTATUS_CONFLICT	Do not remove from the list of conflicts.
%CONFLICTSTATUS_ABSTAIN	Perform the default conflict resolution.

This property is read-write.

ExcludeProperty

Description

This property indicates to synchronization server that the contents of an attachment should *not* be downloaded to the mobile device. (The header is still downloaded.)

This property is only used with attachments.

This property is read-write.

Example

```
If ((%SyncServer.ClientPlatform = %MobileDevice_PDA) And
   (SERVICEORDER_DAMAGEDPART.SIZE > &nMaxAttachmentSize)) Then
   %SyncServer.ExcludeProperty = True;
End-If;
```

ExportLocation

Description

This property allows a developer to instruct the mobile device to export a copy of an attachment to a specific file on the device's disk during synchronization.

You can use this property to automatically extract product images (JPEGs) to a directory on the mobile device to make them accessible from HTML image tags included in HTML that is generated as part of an HTML Area.

This property takes a value of a relative or fully qualified path and filename. If you want to put files into the Res directory (all graphics must be in this directory to display correctly) you could use the following PeopleCode:

```
%SyncServer.ExportLocation = "..\res\filename.ext";
```

This property is used in the OnGetProperty event for an attachment reference file.

This property is read-write.

PropertyValue

Description

This property provides the path and filename of the attachment file on the synchronization server. It is used with the GetAttachment and PutAttachment functions.

In the OnGetProperty event, set **PropertyValue** in the GetAttachment function to the filename and location of the attachment file.

In the OnSetProperty event, the PutAttachment function returns the location of the attachment file in **PropertyValue**.

This property is read-write.

Example

In the code example below, the second **If** statement checks for whether the attachment was successfully retrieved, prior to setting the value for PropertyValue.

```
If QE_MB_CUST_NEED.FTPSITE <> "" And
      QE_MB_CUST_NEED.ATTACHSYSFILENAME <> "" Then
   %SyncServer.PropertyValue = "c:\PSMobile_FileAttach\" |
       QE_MB_CUST_NEED.ATTACHSYSFILENAME;
   &Result = GetAttachment(QE_MB_CUST_NEED.FTPSITE, QE_MB_CUST_⇒
NEED.ATTACHSYSFILENAME,
      %SyncServer.PropertyValue);
End-If;
```

SessionID

Description

This property returns the session ID. This is not unique for a device. It is used for internal chunking in OnValidateSet, and can only be used with this event.

This property is read-only.

See OnValidateSet event.

Example

The following PeopleCode sets validation for batches of instances of the QE_DEMO_ CONTACT Component Interface.

```
SQLExec("UPDATE PSSYNCSERVERVAL SET SYNCSTATUS=0 FROM
PSSYNCSERVERVAL A, %Table(QE_DEMO_CONTACT) B WHERE A.SYNCSESSIONID =
:1 AND A.SYNCTYPEID = :2 AND (A.SYNCID = B.SYNCID AND A.SYNCDTTM >=
B.LASTUPDDTTM)", %SyncServer.SessionID, %SyncServer.TypeID);

SQLExec("UPDATE PSSYNCSERVERVAL SET SYNCSTATUS=1 FROM
PSSYNCSERVERVAL A, %Table(QE_DEMO_CONTACT) B WHERE A.SYNCSESSIONID =
:1 AND A.SYNCTYPEID = :2 AND (A.SYNCID = B.SYNCID AND A.SYNCDTTM <⇒
 B.LASTUPDDTTM)',
%SyncServer.SessionID, %SyncServer.TypeID);
```

Synchronizing

Description

This property is True during synchronization and False otherwise. Use this to perform (or omit) tasks during synchronization. For example, you might limit the number of rows processed in a child scroll during synchronization.

This property is read-only.

SyncIDs

Description

This property is an array containing the synchronization Ids. Populate the SyncIDs array with the sync Ids of the synchronizable Component Interface objects that you want to be synchronized with the mobile objects on the mobile device. Use the Push method to add a SyncID to the SyncIDs array.

The SyncID array is used as a list of sync Ids for instances that satisfy a set of relevance criteria, such as only synchronize Customer objects that are associated to one salesperson. A SQL object that runs a select on a view might be used to calculate the set of sync Ids for Customer objects that are assigned to that salesperson.

This property can be used only with the OnSelect event.

This property is read-write.

See Also

Chapter 7, Array Class , page 633.

TypeID

Description

This property contains the sync ID for the current Component Interface type (not just an instance, but the entire type.) This property is used for internal chunking in OnValidateSet, and can only be used with this event.

This property is read-only.

See LINK I will link with the OnValidateSet event.

Example

The following PeopleCode sets validation for batches of instances of the QE_DEMO_ CONTACT component interface.

```
SQLExec("UPDATE PSSYNCSERVERVAL SET SYNCSTATUS=0 FROM
PSSYNCSERVERVAL A, %Table(QE_DEMO_CONTACT) B WHERE A.SYNCSESSIONID =
:1 AND A.SYNCTYPEID = :2 AND (A.SYNCID = B.SYNCID AND A.SYNCDTTM >=
B.LASTUPDDTTM)", %SyncServer.SessionID, %SyncServer.TypeID);

SQLExec("UPDATE PSSYNCSERVERVAL SET SYNCSTATUS=1 FROM
PSSYNCSERVERVAL A, %Table(QE_DEMO_CONTACT) B WHERE A.SYNCSESSIONID =
:1 AND A.SYNCTYPEID = :2 AND (A.SYNCID = B.SYNCID AND A.SYNCDTTM <⇒
 B.LASTUPDDTTM)', %SyncServer.SessionID, %SyncServer.TypeID);
```

validateID

Description

This property contains the sync ID for the only current *instance* of the Component Interface. It is created using the DateTime field for that component interface (that field is set in the record properties for that Component Interface.)

This property can be used only with the OnValidate event.

This property is read-write.

validateRowCount

Description

This property returns the total number of rows in a hierarchical synchronizable Component Interface object.

For example, a service order with 10 lines has a validateRowCount value of 11. The property default is 1.

This property can be used only with the OnValidate event.

This property is read-only.

validateVersion

Description

This property contains the version for the instance of the Component Interface. Set validateVersion using the SYNCDTTM field. If the Component Interface object does not exist, or if the object is not to be used for some other reason (such as security), then do not set the validateVersion property.

This property can be used only with the OnValidate event.

This property is read-write.

Advanced Considerations for Synchronization

After you mobile application is running without errors, you might decide to further tune your application. Two events enable you to do this in specific circumstances.

This section discusses the following:

- Using OnValidateSet
- Using OnGetDefinition

Using OnValidateSet

You should only use OnValidateSet when you have a large number of Component Interface instances to be synchronized, more than 1000.

Before you try to create an OnValidateSet PeopleCode program, you should first create OnValidate PeopleCode. This enables you to determine the shape of the Component Interface instance, the number of child rows and collections. Having a thorough understanding of this helps you create the OnValidateSet PeopleCode.

Though the OnValidateSet PeopleCode may improve performance, there is additional maintenance with it. Every time you add an exposed collection, or remove one, you must change your PeopleCode program to reflect the new data structure.

Note. If you have any PeopleCode in OnValidateSet, including a program that has been commented out, the OnValidate event does not run.

When this event occurs, a table on the synchronization server (PSSYNCSERVERAL) is populated with all of the synchronization Ids for the component type. They table is keyed by Session ID (SYNCSESSIONID field) and Type ID (SYNCTYPEID field).

The status of the row is tracked in the status column (SYNCSTATUS field). All of the rows in the table begin with a status of 3, meaning that the corresponding Component Interface instance should be purged from the synchronization server. The other valid values for the status field are:

Status	Description
0	Current. This causes the corresponding Component Interface instance to not be synchronized.
1	Out of date. This causes a **Get** to be called on the corresponding Component Interface instance, and the rest of the event flow to be joined.

The OnValidateSet event uses the SyncServer properties **SessionID** and **TypeID**. These should be populated by values from the SYNCSESSIONID and SYNCTYPEID fields, respectively.

Considerations Using OnValidateSet

If you set breakpoints in a Component Interface, or do a trace , you will see OnValidateSet running multiple times. This happens when the synchronization server sends a request to an application server for OnValidateSet to run and does not receive a response soon afterward. Then the synchronization server sends out another request. In theory, OnValidateSet only runs once per Component Interface Type. In practice, more than one OnValidateSet request runs concurrently in certain environments.

OnValidateSet PeopleCode Example

The following PeopleCode sets validation for batches of instances of the QE_DEMO_ CONTACT component interface.

```
%SyncServer.AddReferencedType(CompIntfc.RefCI);
```

Using OnGetDefinition

Using the AddReference and AddReferenceType methods, you can create ad-hoc references to Component Interfaces using PeopleCode, instead of defining the reference and the relationship in Application Designer.

Use the OnGetDefinition event when you want to access the ad-hoc references specified with the AddReference method in the component events. In this event, you must specify every instance of an ad-hoc reference. The referenced Component Interfaces are defined by their SyncIDs. The referenced Component Interfaces objects can reference other referenced Component Interface objects.

When to Use OnGetDefinition

You should only consider using ad-hoc references when you want to dynamically establish read-only references of static data. Using these methods, you are not establishing a relationship between the Component Interfaces. You cannot navigate these references. You cannot make any changes to these instances.

OnGetDefinition PeopleCode Example

In the OnGetDefinition event, have one call to the AddReferenceType method for every referenced Component Interface type in the current Component Interface.

```
%SyncServer.AddReference(&SyncID, CompIntfc.RefCI);
```

In one of the Component build events (PreBuild, PostBuild, RowInit) you must also include a call to the AddReference method for every instance (identified by the SyncID) of the referenced Component Interface.

```
SQLExec("UPDATE PSSYNCSERVERVAL SET SYNCSTATUS=0
FROM PSSYNCSERVERVAL A, %Table(QE_DEMO_CONTACT) B
WHERE A.SYNCSESSIONID =:1 AND A.SYNCTYPEID = :2
AND (A.SYNCID = B.SYNCID AND A.SYNCDTTM >= B.LASTUPDDTTM)",
%SyncServer.SessionID, %SyncServer.TypeID);

SQLExec("UPDATE PSSYNCSERVERVAL SET SYNCSTATUS=1
FROM PSSYNCSERVERVAL A, %Table(QE_DEMO_CONTACT) B
WHERE A.SYNCSESSIONID = :1 AND A.SYNCTYPEID = :2
AND (A.SYNCID = B.SYNCID AND A.SYNCDTTM < B.LASTUPDDTTM)’,
%SyncServer.SessionID, %SyncServer.TypeID);
```

CHAPTER 33

Tree Classes

This chapter provides an overview of Tree classes and discusses the following topics:

- Relationships between different Tree classes.
- Collections in the Tree classes.
- Error handling with trees.
- Leaves and nodes insert verification.
- Data types for tree objects.
- Scope of the tree objects.
- Tree classes implementation.
- Tree classes reference.

Understanding Tree Classes

Using the Tree classes in your PeopleCode, you have access to all the functionality of PeopleSoft Tree Manager. Your application should instantiate the appropriate tree objects when it must work with the tree system database data, call the appropriate methods and properties, then close the objects when it is finished.

Creating or deleting a tree *object* does not create or delete tree system database information. You must call the method for that tree object directly to create or delete database information, that is, the **Create** or **Delete** method.

One instance of a tree or tree structure object can be created to work on multiple database entities. However, only one tree or tree structure can be open at a time. If you open a Tree before closing the one that's currently open, you receive an error. You must explicitly close a tree or tree structure (using the **Close** method) before you try to open a second one.

All of the classes, and most of the properties and methods that make up the Tree Classes have a GUI representation in PeopleSoft Tree Manager. This document assumes that the reader is familiar with PeopleSoft Tree Manager.

The following classes make up the Tree classes:

- Branch Collection
- Leaf
- Level
- Level Collection

- Node
- Tree
- Tree Collection
- Tree Structure
- Tree Structure Collection

With most of the classes of objects in PeopleTools, when you use a **Getxxx** method, you are fully instantiating an object. However, for the tree class, when you use **GetTree** (from the Session object), you get a closed tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method. Working with closed trees can improve your performance. The same applies to a tree structure: it's closed when you get it from the Session object, and you must open it before you can access its properties or change it.

There aren't any built-in functions for the Tree classes: objects are instantiated from identifiers, from other objects, or from a Session object.

You can use the TreeDetailInNode function to determine whether a specific record field value is a descendant of a specified node in a specified tree.

Using the GenerateTree function, you can produce a GUI representation of a tree in the PeopleSoft Internet Architecture.

See Also

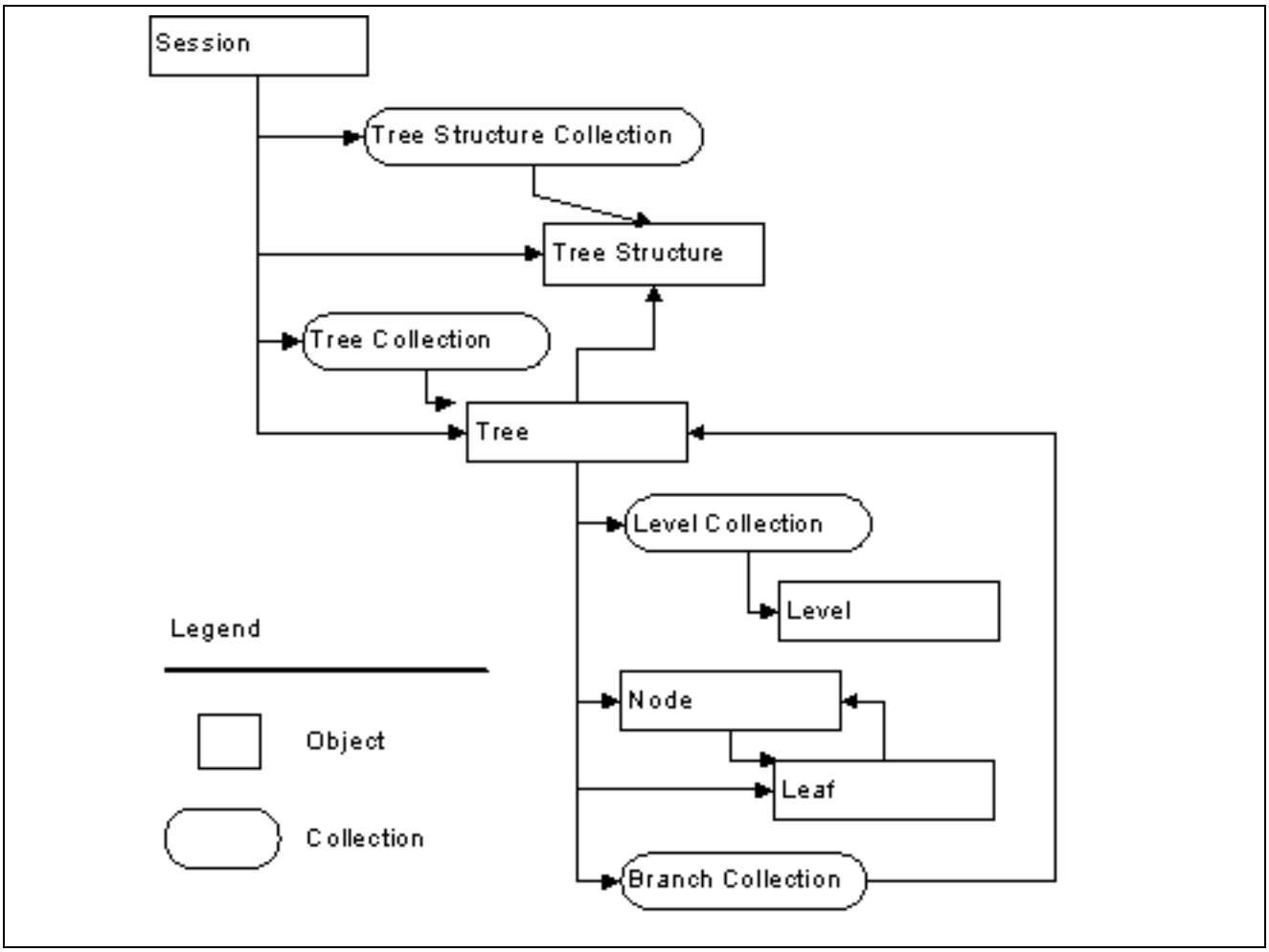
Chapter 1, PeopleCode Built-in Functions, TreeDetailInNode , page 480

Chapter 29, Session Class , page 1559

PeopleCode Developer's Guide, Using Methods and Built-In Functions, Using the GenerateTree Function

Relationships Between Different Tree Classes

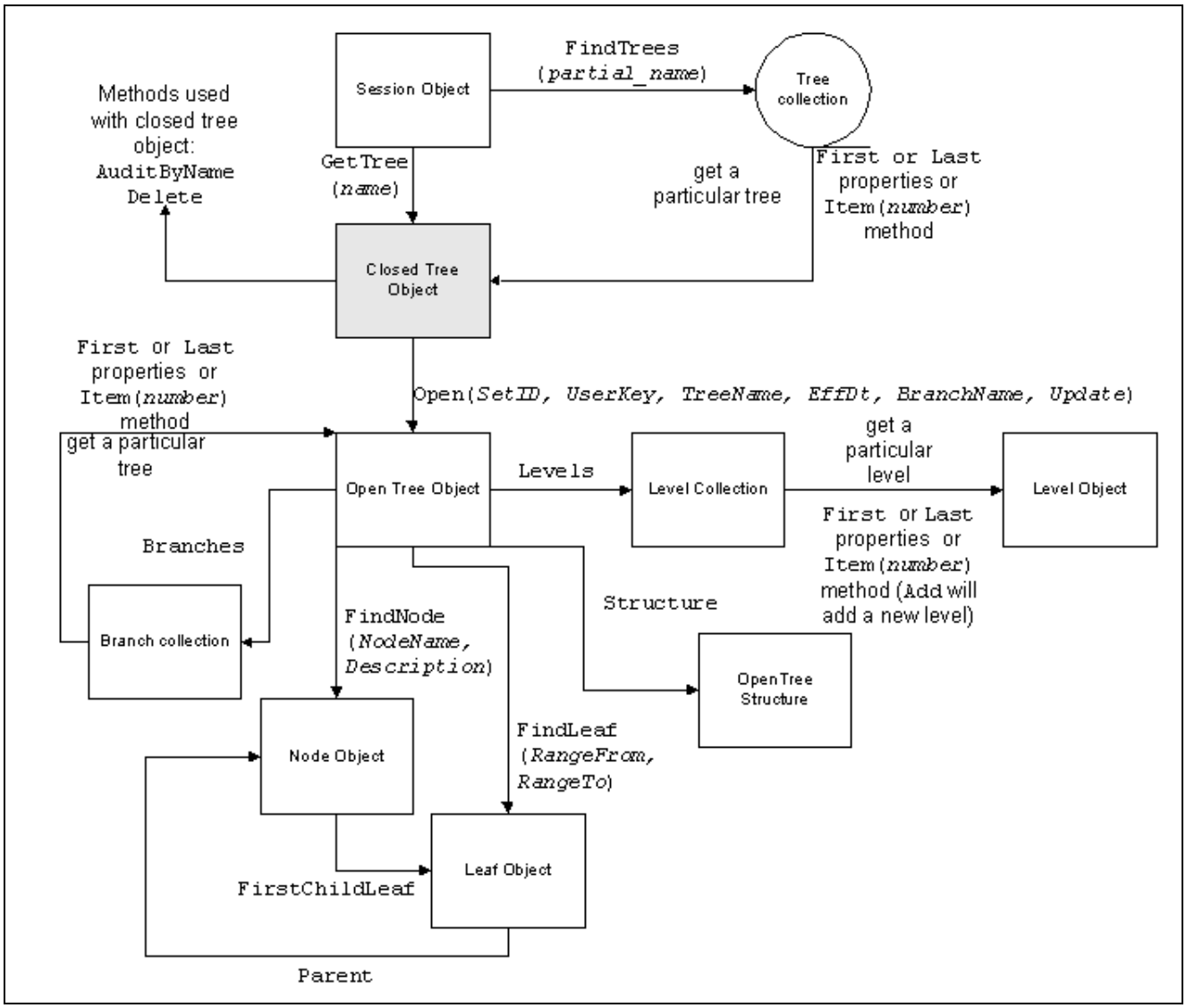
The following diagram shows the Tree classes and their relationships. Objects at the tip of the arrowhead are accessed or created from the object at the left of the arrowhead. Squares indicate objects. Ovals indicate collections.



Tree Classes and their relationships

- From the session object, you can get identifiers for a tree or a tree structure, or you can instantiate a tree collection or a tree structure collection.
- From the tree collection object, you can get an identifier for a tree.
- From the tree structure collection object, you can get an identifier for a tree structure, or you can instantiate a tree structure object.
- From the level collection object, you can instantiate a level object.
- From the branch collection object, you can get an identifier for a tree.
- From the tree object, you can instantiate a tree, a tree structure, a branch collection, a leaf, a node, and a level collection.
- From the node object, you can instantiate a leaf object or a node object.
- From the leaf object, you can instantiate a leaf object or a node object.

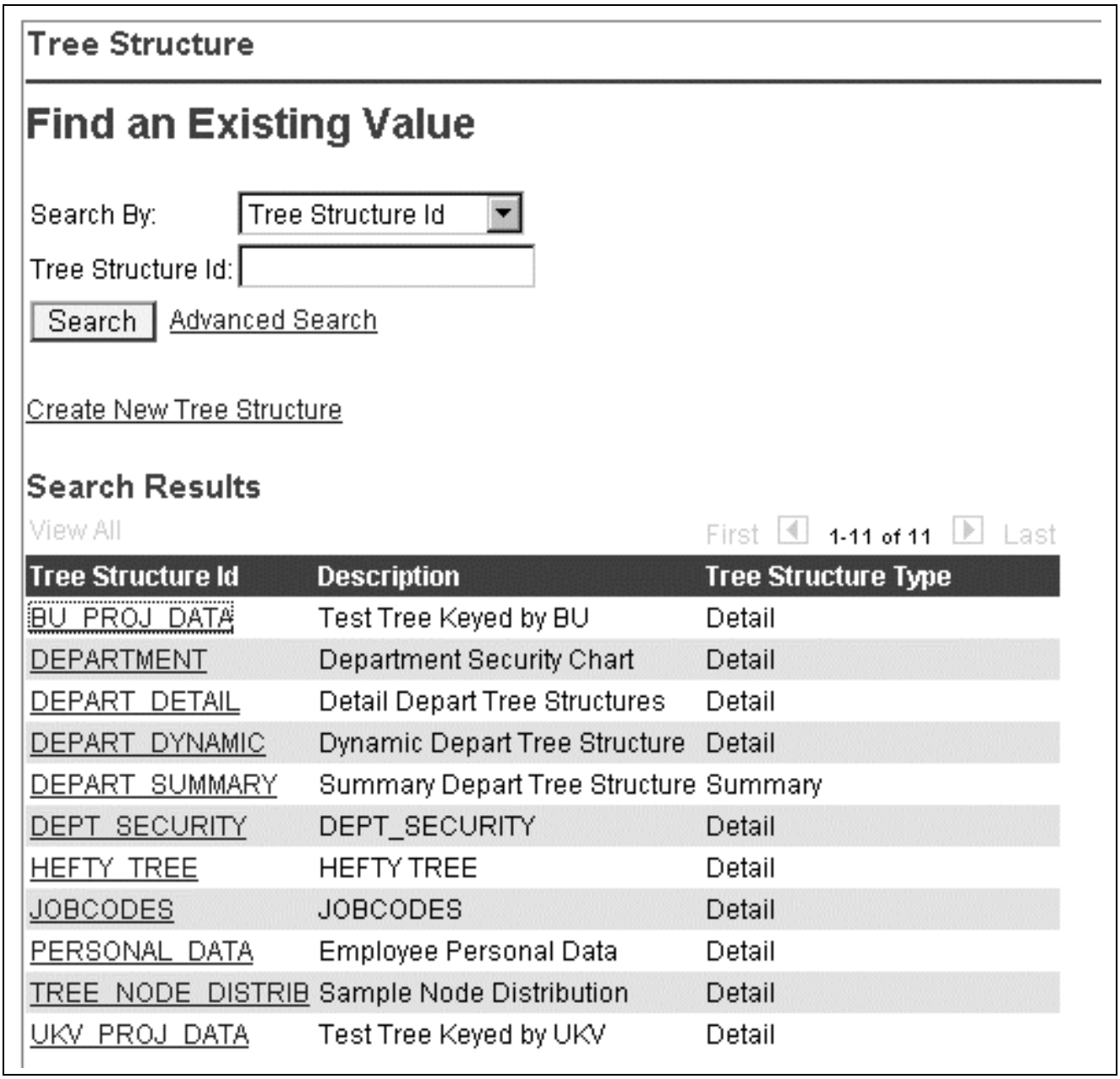
Here’s another diagram, listing the different tree classes, with one of the properties or methods used for instantiating one class from another.



Tree Classes flow with methods or properties

Collections in the Tree Classes

A *collection* is a set of similar things, like a group of already existing tree structures or trees. Like everything else in the tree classes, collections have a GUI representation. For example, when you find an existing tree structure from PeopleSoft Tree Manager, you get a dialog that lets you select the tree structure you want. This dialog box data is represented in PeopleCode as the *tree structure collection.*



Collection of tree structures

The following collections are part of the Tree classes:

- Branch collection
- Level collection
- Tree collection
- Tree structure collection

Error Handling With Trees

The Tree classes log descriptive information regarding errors and warnings to the PSMessages collection, instantiated from a session object.

In your program, use the PSMessages collection to identify and report to the user any errors that are encountered during processing.

The tree classes log errors "interactively", that is, as they happen. For example, suppose you had created a new tree, and were setting the effective date using the effective date property (**EffDt**). If you used an invalid value for the effective date, the error would be logged in the PSMessages collection as soon as you set the value, not when you saved the tree.

When you want to check for errors depends on your application. However, if you check for errors after every assignment, you may see a performance degradation.

One way to check for errors is to check the number of messages in the PSMessages collection, using the **Count** property. If the Count is 0, no error or warning messages have been logged to the PSMessages collection.

```
Local ApiObject &MYSESSION;
Local ApiObject &ERRORCOL;
Local ApiObject &TREE, TREELIST;

&MYSESSION = %Session;

If &MYSESSION Then
   /* connection is good */

   &TREELIST = &MYSESSION.FindTree("", "", "DEPARTMENT", "", "");

   For &I = 1 to &TREELIST.Count
      &TREE = &TREELIST.Item(&I);

   /* Do processing */

      &TREE.Save();

      /* Do error checking */

   &ERRORCOL = &MYSESSION.PSMessages;
   If (&ERRORCOL.Count <> 0) Then
      /* errors occurred – do processing */
   Else
      /* no errors */
   End-If;
   End-For;
Else
   /* do processing for no connection */
End-If;
```

See Also

Chapter 29, Session Class , Error Handling , page 1560

Leaves and Nodes Insert Verification

InsertChildLeaf, InsertSibNode, InsertChildNode, and so on, can return False or a Null object reference, depending upon the type of error encountered. You may want to declare references to new leaves or nodes as type ANY until after you verify they were actually created. You can do this by using the None or All PeopleCode functions to determine whether a valid Leaf or Node Object was created. If a Leaf or Node object was created ALL()returns True and None() returns False.

```
&NewLeaf = &RootNode.InsertChildLeaf("8000", "8999");
If NONE(&NewLeaf) Then
   /* Leaf not inserted, do error processing */
   &Messages = &Session.PSMessages;
   If &Messages.WarningPending Then
      /* do error processing */
   End-if;
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, None , page 335

Chapter 1, PeopleCode Built-in Functions, All , page 41

Data Types for Tree Objects

All tree objects, that is, trees, tree structures, nodes, levels, and so on, are declared as type ApiObject. For example:

```
Local ApiObject &MYTREE;

Global ApiObject &MYNODE;
```

All tree objects can be declared as Local, Component or Global.

Scope of the Tree Objects

All tree objects, that is, trees, tree structures, nodes, leaves, and so on, can *only* be instantiated from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

You can instantiate a tree or tree structure object only from a session object. You have to instantiate the session object, and connect to the database, before you can instantiate a tree or tree structure.

```
Local ApiObject &TREELIST;
Local ApiObject &MYSESSION;

&MYSESSION = %Session;

If &MYSESSION Then
   /* connection is good */
Else
   /* do error processing */
End-if;
```

Note. Tree Classes are accessible only through PeopleCode.

Tree Classes Implementation

You will often want to create a new tree. The following procedure discusses this action in more detail.

The TreeMover Application Engine program uses the Tree API (and File Layouts) for importing a tree from a flat file or exporting a tree to a flat file.

To create a new tree:

In this example, you are creating a new tree based on an existing tree structure. The following is the complete code sample: the steps explain each line.

```
Local ApiObject &Session;
Local ApiObject &TreeList, &MyTree;
Local ApiObject &LvlColl;

&Session = %Session;

&TreeList = &Session.findtree("", "", "", "", "");

If ALL(&TreeList) Then
   &MyTree = &TreeList.first;

   /* create new tree */
   &TreeReturn = &MyTree.Create("", "", "PERSONAL_DATA2", "1999-06-01", ⇒
"PERSONAL_DATA");

   &MyTree.description = "test tree";

   /* add level */
   &LvlColl = &MyTree.levels;
   &Level = &LvlColl.add("FIRST LVL");
   &Level.description = "First Level";

   /* add node */
```

```
   &MyTree.insertroot("00001");

   /* add new leaf */
   &RootNode = &MyTree.FindRoot();

   If ALL(&RootNode) Then
   &NewLeaf = &RootNode.InsertChildLeaf("8000", "8999");

   /* save new tree */
   &RSLT = &MyTree.Save();

   /* Do error checking */
   If Not (&RSLT) Then
      /* errors occurred = do error checking */
      &ERRORCOL = &Session.PSMessages;
      For &I = 1 To &ERRORCOL.count
         /* do error processing */
      End-For;
   Else
      /* no errors - saved correctly - do other processing */
   End-If;
   End-if;
End-if;
```

1. Get a session object.

   Before you can get a tree, you have to get a session object. The session controls access to the tree, provides error tracing, enables you to set the runtime environment, and so on. Use the %Session system variable to return a reference to the current PeopleSoft session.

   ```
   &Session = %Session;
   ```

2. Get a tree object.

   The **FindTree** method specified with no parameters returns a collection of all the trees already created. To ensure you have a valid tree list, use the ALL built-in function. Because we're creating a new tree, it doesn't matter which tree we get originally, so use the **First** method to get the first tree in the collection.

   ```
   &TreeList = &Session.findtree("", "", "", "", "");

   If ALL(&TreeList) Then
      &MyTree = &TreeList.First;
   ```

   After you execute the **First** method, you have only the **structure** of the tree, that is, a closed tree. You haven't populated the tree with data yet.

3. Create the Tree.

   The **Create** method creates a new tree with the name PERSONAL_DATA2. Description is a required property (if you don't specify something for Description you cannot save the tree.)

   ```
   &TreeReturn = &MyTree.Create("", "", "PERSONAL_DATA2", "1999-06-01", ⇒
   "PERSONAL_DATA");
   ```

```
&MyTree.description = "test tree";
```

4. Add a level.

   To add a level, you have to instantiate a level collection. Although there aren't any levels in the tree, you can still access this collection. Use the **Add** method with the level collection to add a new level. Remember, the level name *must be 8 characters or less*. Description is a required property (if you don't specify something for Description you cannot save the tree.)

   ```
   &LvlColl = &MyTree.levels;
   &Level = &LvlColl.add("FIRST LVL");
   &Level.description = "First Level";
   ```

5. Add the root node.

   Because this is a new tree, you must first add the root node.

   ```
   &MyTree.insertroot("00001");
   ```

6. Add a leaf.

   To add a new leaf, you must return a reference to the node object. Using the All built-in function ensures that there is a root node before you try to insert the leaf with the **InsertChildLeaf** method.

   ```
   &RootNode = &MyTree.FindRoot();

   If ALL(&RootNode) Then
      &NewLeaf = &RootNode.InsertChildLeaf("8000", "8999");
   ```

7. Save the tree.

   When you execute the **Save** method, the new tree is saved to the database.

   ```
   &RSLT = &MyTree.Save();
   ```

   The **Save** method returns a Boolean value: True if the save was successful, False otherwise. You can use this value to do error checking.

   Note. If you're running the tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

   ---

8. Check for errors.

   You can check if there were any errors using the PSMessages property on the session object.

   ```
   If Not (&RSLT) Then
      /* errors occurred = do error checking */
      &ERRORCOL = &Session.PSMessages;
      For &I = 1 To &ERRORCOL.count
         /* do error processing */
      End-For;
   Else
      /* no errors - saved correctly - do other processing */
   End-If;
   ```

If there are multiple errors, all errors are logged to the PSMessages collection, not just the first occurrence of an error. As you correct each error, delete it from the PSMessages collection.

Note. If you've called the Tree API from an Application Engine program, all errors are also logged in the AE error log tables.

Session Class Methods

Tree Classes don't have any built-in functions. Instead, they're instantiated from the Session Class.

In this section, we discuss the Session class methods. The methods are discussed in alphabetical order.

FindTree

Syntax

FindTree(*SetId, UserKeyValue, TreeName, EffDt, BranchName*)

Description

The **FindTree** method returns a tree collection.

You can use wildcards or partial keys with any of the parameters to get a smaller subset of the tree collection. You can use a percent symbol (%) in place of a number of characters. You can use an underscore (_) in place of one character. You can also use a partial value for any key. For example, to get a collection of all the trees whose names started with the letter "B", you could specify Null strings for all the parameters (that is, two quotation marks with no space between them ("")), and specify just the letter B for the *treename*:

```
&MYTREECOL = &MYSESSION.FindTree("", "", "B", "", "");
```

Considerations Using Effective Dates

Consider the following with using effective dates:

- When you specify a value for *EffDt*, all trees with dates less than or equal to the date specified by *EffDt* are returned.
- If you use a value for *EffDt* that doesn't match any trees in the database, this method returns a 0.
- If you specify an asterisks ("*") for *EffDt* and no other parameters, the list of returned trees is sorted by effective date, with the maximum effective date tree listed first. After the tree list is sorted by EffDt, it's sorted by SetID, user key value, tree name, then branch name.
- If you specify the name of a tree, the list of the returned trees is sorted by effective date, with the maximum effective-date tree listed first.

Parameters

Parameter	Description
SetID	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
UserKeyValue	Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
TreeName	Specify the name of the tree to find. This parameter takes a string value.
EffDt	Specify the effective date for the tree to find. This parameter takes a string value.
BranchName	Specify the name of the branch for the tree to find. This parameter takes a string value. If the tree is unbranched, you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

A tree collection object. If no tree collection object matching the parameters is found, it returns a 0.

Example

The following code populates a tree collection object with a list of all the trees named "DEPARTMENT".

```
Local ApiObject &TREELIST;
Local ApiObject &MYSESSION;

&MYSESSION = %Session;

If &MYSESSION Then
   /* connection is good */
   &TREELIST = &SESSION.FindTree("", "", "DEPARTMENT", "", "");
Else
   /* do error processing */
End-if;
```

The collection &TREELIST includes all trees that have the name "DEPARTMENT, regardless of SetID, Effective Date, and so on.

FindTreeStructure

Syntax

FindTreeStructure(*StructId*)

Description

The **FindTreeStructure** method returns a tree structure collection. *StructId* is a string value.

You can use wildcards or partial keys with *StructId* to get a subset of the list of tree structures. Use a percent sign (%) in place of a number of characters. Use an underscore (_) in place of one character. You can also use a partial value for *StructId*. For example, to get a collection of all the tree structures whose names started with the letter "B", you could specify just the letter B:

```
&MYTREESCOLL = &MYSESSSION.FindTreeStructure("B");
```

Returns

A tree structure collection.

Example

```
Local ApiObject &TLIST;
Local ApiObject &MYSESSION;

&MYSESSION = %Session;

If &MYSESSION Then
   /* connection is good */
   &TSLIST = &MYSESSION.FindTreeStructure("DEPT");
Else
   /* do error processing */
End-if;
```

GetTree

Syntax

GetTree(*SetID*, *UserKeyValue*, *TreeName*, *EffDt*, *BranchName*)

Description

The **GetTree** method returns a closed tree object. Before you can use some of the methods or any of the properties, you must **Open** a tree object.

Note. You *cannot*use any of the key properties (**KeyName**, **KeySetId**, and so on) with the closed tree object returned by **GetTree**. You can use these properties only with the trees returned in a collection by one of the Session **Find** methods.

Parameters

Although all the parameters for GetTree are required, they are unused. You should specify a Null string (that is, two quotation marks with no blank space between them ("")) for all the parameters.

Returns

A reference to a closed tree object.

Example

To create a new tree, use the following:

```
&MYTREE = %Session.GetTree("", "", "", "", "");
```

```
&MYTREE.Create("","","PERSONAL_NEW", "05-05-1997", "PERSONAL_DATA");
```

To determine if a tree already exists in the database, use the following code:

```
&MYTREE = %Session.GetTree("", "", "", "", "");
If &MYTREE.Exists("","","PERSONAL_OLD", "05-05-1997", "PERSONAL_DATA") = 0 then
     /* Do processing */
End-If;
```

To open an existing tree, use the following code:

```
&MYTREE = %Session.GetTree("", "", "", "", "");
&MYTREE.Open("","","PERSONAL_OLD", "05-05-1997", "PERSONAL_DATA", True);
```

GetTreeStructure

Syntax

GetTreeStructure(*StructId*)

Description

The **GetTreeStructure** method returns a closed tree structure object. *StructId* is a string value. Before you can use some of the methods or any of the properties, you must **Open** the tree structure object.

Branch Collection

A branch collection is returned from the Branches property, used with an open tree object.

```
&MYBRANCHCOLL = &MYTREE.Branches;
```

&MYTREE must be a branched tree. To verify whether a tree is branched, you can check the value of the IsBranched property of a tree object (Boolean value: True or False.)

Most of the branch collection methods return a *closed* tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method. Working with closed trees can improve your performance.

See Chapter 33, Tree Classes , Branches, page 1724.

Branch Collection Method

In this section, we discuss the Item method.

Item

Syntax

Item(*SetID*, *UserKeyValue*, *TreeName*, *EffDt* , *BranchName*)

Description

The **Item** method returns the specified branch in the branch collection object executing the method as a *closed* tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method. The branch must be part of the branch collection executing the method.

Parameters

Parameter	Description
SetID	Specify the table indirection key for the branch (tree). This parameter takes a string value. If the associated tree structure has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
UserKeyValue	Specify the User Key Value for the branch (tree). This parameter takes a string value. If the associated tree structure has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
TreeName	Specify the name for the tree the branch is in. This parameter takes a string value.
EffDt	Specify the effective date for this branch. This parameter takes a string value.
BranchName	Specify the name of the branch for the tree. This parameter takes a string value.

Example

```
&MYTREE = &MYBRANCHCOLL.Item("", "", "PERSONAL_DATA", "05-05-1997", "");
```

Branch Collection Properties

In this section, we discuss the Branch collection properties. The properties are discussed in alphabetical order.

Count

Description

Returns the number of branches for the branch collection object.

First

Description

The **First** property returns the first branch in the branch collection object executing the method as a *closed* tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

Last

Description

The **Last** property returns the last branch in the branch collection object executing the method as a *closed* tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

Next

Description

The **Next** property returns the next branch in the branch collection object executing the method as a *closed* tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

You can use this method only after you have used either the **First** or **Item** methods: otherwise the system doesn't know where to start.

Leaf Class

Leaf objects are instantiated from other tree classes as follows:

- From a tree object with the FindLeaf method.
- From a node object with the FirstChildLeaf property.
- From another leaf object with the NextSib and PrevSib properties.

See Chapter 33, Tree Classes , FindLeaf, page 1710; Chapter 33, Tree Classes , FirstChildLeaf, page 1696; Chapter 33, Tree Classes , NextSib, page 1676 and Chapter 33, Tree Classes , PrevSib, page 1676.

Leaf Class Methods

In this section, we discuss the Leaf class methods. The methods are discussed in alphabetical order.

Cut

Syntax

```
Cut()
```

Description

The **Cut** method cuts the leaf and puts it into a buffer (the clipboard.) You can then use the PasteSib method to add the leaf back as a sibling of a different leaf. You can also use the PasteChild node method to add the leaf back as a child of a different node.

You can only have one object at a time on the clipboard. If you cut another leaf or node, the leaf in the clipboard is overwritten.

Parameters

None.

Returns

None.

See Also

Chapter 33, Tree Classes , Cut, page 1684, Chapter 33, Tree Classes , PasteSib, page 1673, Chapter 33, Tree Classes , PasteChild, page 1692.

Delete

Syntax

Delete()

Description

The **Delete** method deletes the leaf object executing the method *from the database*.

Example

```
&MYLEAF = &MYTREE.FindLeaf("8200", "8300");
&MYLEAF.Delete();
```

DeleteByRange

Syntax

DeleteByRange(*RangeFrom*, *RangeTo*)

Description

The **DeleteByRange** method deletes the specified leaf object from the database. The leaf specified by the parameters does *not* have to match the leaf executing the method. That is, if &MYLEAF is associated with a range 8200-8300, you can specify a different range with the **DeleteByRange** method. For example:

```
&MYLEAF = &MYTREE.FindLeaf("8200", "8300");
/* some processing */
&MYLEAF.DeleteByRange("8400", "8500");
```

Parameters

Parameter	Description
RangeFrom	Specify the starting range of the leaf to be deleted. This parameter takes a string value.
RangeTo	Specify the ending range of the leaf to be deleted. This parameter takes a string value.

Example

```
&MYLEAF.DeleteByRange("8400", "8500");
```

InsertDynSib

Syntax

InsertDynSib()

Description

The **InsertDynSib** method inserts a dynamic leaf as a sibling leaf to the leaf object executing the method. The leaf is a *new*node.

A leaf object associated with the new leaf is returned. The new leaf is inserted as the next sibling leaf. If the new leaf isn't inserted successfully, the method returns Null.

Note. To insert a sibling leaf with a specific range, use the **InsertSib** method.

Parameters

None.

Returns

A reference to the new leaf. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted is to test the result using either the All or None functions.

Example

```
&NEWLEAF = &MYLEAF.InsertDynSib();
```

See Also

Chapter 33, Tree Classes , InsertSib, page 1670, Chapter 1, PeopleCode Built-in Functions, All , page 41, Chapter 1, PeopleCode Built-in Functions, None , page 335.

InsertSib

Syntax

InsertSib(*RangeFrom*, *RangeTo*)

Description

The **InsertSib** method inserts a new leaf as a sibling leaf under the leaf currently executing the method. The leaf specified by the range parameters must be a *new* leaf. You receive an error if the leaf already exists.

A leaf object associated with the new leaf is returned. The new leaf is inserted as the next sibling leaf, although there is no explicit ordering of leaves: leaves take the order of the alphanumeric database sort on their range fields.

Note. To insert a dynamic sibling leaf (that is, without specifying a range) use the **InsertDynSib** method.

Parameters

Parameter	Description
RangeFrom	Specify the starting range of the new leaf. This parameter takes a string value.
RangeTo	Specify the ending range of the new leaf. This parameter takes a string value.

Returns

A leaf object associated with the new leaf. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted is to test the result using either the All or None functions.

Example

```
&NEWLEAF = &MYLEAF.InsertSib("10090", "10100");
```

See Also

Chapter 33, Tree Classes , InsertDynSib, page 1670, Chapter 1, PeopleCode Built-in Functions, All , page 41, Chapter 1, PeopleCode Built-in Functions, None , page 335.

MoveAsChild

Syntax

MoveAsChild(*Node*)

Description

The **MoveAsChild** method moves the leaf executing the method to a different node in the tree. The leaf becomes the first child leaf under the node, even though there is no explicit ordering of leaves: leaves take the order of the alphanumeric database sort on their range fields. The specified node becomes the new parent to the leaf.

Parameters

Parameter	Description
Node	Specify a node object. This parameter value must be an object, not a string or a name.

Note. To specify a node name, not a node object, use the **MoveAsChildByName** method.

Example

The following example moves leaf 8001 from node 10100 (old parent) to node 00001 (new parent)

```
&MY_LEAF = &MY_TREE.FindLeaf("8001", "8001");
&NEW_PARENT = &MY_TREE.FindNode("00001", "");
If &NEW_PARENT <> Null Then
   &MY_LEAF.MoveAsChild(&NEW_PARENT);
End-If;
```

See Also

Chapter 33, Tree Classes , MoveAsChildByName, page 1672.

MoveAsChildByName

Syntax

MoveAsChildByName(*NodeName*)

Description

The **MoveAsChildByName** method moves the leaf executing the method to a different node in the tree. The leaf becomes the first child leaf under the node, although there is no explicit ordering of leaves: leaves take the order of the alphanumeric database sort on their range fields. The specified node becomes the new parent to the leaf. The node specified by *NodeName* must be a valid node name.

Parameters

Parameter	Description
NodeName	Specify the name of a node. *NodeName* must be a valid node for the existing tree. This parameter takes a string value.

Note. To specify a node object, not a node name, use the **MoveAsChild** method.

Example

The following example moves leaf 8001 from node 10100 (old parent) to node 00001 (new parent)

```
&MY_LEAF = &MY_TREE.FindLeaf("8001", "8001");
If &MY_LEAF <> Null Then
   &MY_LEAF.MoveAsChildByName("00001");
End-If;
```

See Also

Chapter 33, Tree Classes , MoveAsChild, page 1671.

MoveAsSib

Syntax

MoveAsSib(*Leaf*)

Description

The **MoveAsSib** method moves the leaf executing the method to a new place in the tree. The current leaf becomes the next sibling leaf under the specified leaf object, although there is no explicit ordering of leaves: leaves take the order of the alphanumeric database sort on their range fields.

Parameters

Parameter	Description
Leaf	Specify a leaf object. This parameter value must be an object, not a string or a name.

Note. To specify a range, not a leaf object, use the **MoveAsSibByRange** method.

Example

```
&MYLEAF = &MYTREE.FindLeaf("8000", "8000");
&MYLEAF2 = &MYTREE.FindLeaf("9000", "9000");
&MYLEAF.MoveAsSib(&MYLEAF2);
```

See Also

Chapter 33, Tree Classes , MoveAsSibByRange, page 1673.

MoveAsSibByRange

Syntax

MoveAsSibByRange(*RangeFrom, RangeTo*)

Description

The **MoveAsSibByRange** method moves the leaf executing the method to a new place in the tree. The current leaf becomes the next sibling leaf under the specified leaf object.

Parameters

Parameter	Description
RangeFrom	Specify the starting range of the leaf that you want as the parent of the leaf executing the method. This parameter takes a string value.
RangeTo	Specify the ending range of the leaf that you want as the parent of the leaf executing the method. This parameter takes a string value.

Note. To specify a leaf object, not a range, use the **MoveAsSib** method.

Example

```
&MYLEAF = &MYTREE.FindLeaf("8000", "8000");
&MYLEAF.MoveAsSibByRange("9000", "9000");
```

See Also

Chapter 33, Tree Classes , MoveAsSib, page 1672.

PasteSib

Syntax

PasteSib()

Description

The **PasteSib** method makes the leaf from the buffer (clipboard) a sibling to the leaf executing the method. You must use the **Cut** leaf method before you can paste a leaf.

You can have only one object at a time on the clipboard. If you cut another leaf or node, the leaf in the clipboard is overwritten.

Parameters

None.

Returns

None.

See Also

Chapter 33, Tree Classes , Cut, page 1668 Chapter 33, Tree Classes , PasteChild, page 1692
Chapter 33, Tree Classes , Cut, page 1684 node method

Leaf Class Properties

In this section, we discuss the Leaf class properties. The properties are discussed in alphabetical order.

DisplayLevelNumber

Description

When using the Tree API to create a graphic representation of the tree (such as for HTML Tree Manager or other application pages) use this property to indicate the *display* level, that is, tell the user how many levels deep the leaf is. This is generally used for trees where levels aren't used and the LevelNumber property always returns 0.

This property is read-only.

See Also

Chapter 33, Tree Classes , LevelNumber, page 1699.

Dynamic

Description

This property specifies whether the leaf has a dynamic range or a specified range. If you set this property to True, the leaf has a dynamic range. If this is a new leaf, and you do not set this property, the value is automatically set is False.

This property is read-write.

See Also

.

HasNextSib

Description

This property returns True if the leaf has a next sibling, that is, it isn't the last leaf listed under the parent node.

This property is read-only.

HasPrevSib

Description

This property returns True if the leaf has a previous sibling, that is, it isn't the first leaf listed under the parent node.

This property is read-only.

IsChanged

Description

This property returns True if the leaf has been edited or changed but the tree containing the leaf hasn't been saved.

This property is read-only.

Example

```
If &MYLEAF.IsChanged Then
   &MYTREE.Save();
End-If;
```

IsCut

Description

This property returns True if the leaf has been cut from the displayed tree. This property is generally used with the HTML Tree Manager.

This property is read-only.

IsDeleted

Description

This property returns True if the leaf has been deleted from the tree but the tree hasn't been saved.

This property is read-only.

IsInserted

Description

This property returns True if the leaf has been inserted as a new leaf in the tree but the tree hasn't been saved.

This property is read-only.

NextSib

Description

This property returns a leaf object associated with the next sibling leaf. The next sibling of a leaf is the leaf appearing under the current leaf. If there is no next sibling and you try to assign it to a variable, you receive a runtime error.

This property is read-only.

Example

The following code traversed the leaves from top to bottom.

```
While &MYLEAF.HasNextSib
   &MYLEAF = &MYLEAF.NextSib;
   /* do some processing */
End-While;
```

Parent

Description

This property returns a node object associated with the parent node for the leaf.

This property is read-only.

Example

```
&PARENTNODE = &MYLEAF.Parent;
/* do node processing with node object */
```

PrevSib

Description

This property returns a leaf object associated with the previous sibling leaf. The previous sibling of a leaf is the leaf appearing above the current leaf. If there is no previous sibling and you try to assign it to a variable, you receive a runtime error.

This property is read-only.

Example

The following code traverses the leaves from bottom to top.

```
While &MYLEAF.HasPrevSib
   &MYLEAF = &MYLEAF.PrevSib;
   /* do some processing */
End-While;
```

RangeFrom

Description

This property returns the starting range, as a string, of the leaf.

This property is read-write.

RangeTo

Description

This property returns the ending range, as a string, of the leaf.

This property is read-write.

TreeBranchName

Description

This property returns the branch name of the tree as a string if the tree is branched. If not branched, this property returns a blank string.

This property is read-only.

TreeEffDt

Description

This property returns the effective date of the tree as a string if the tree is effective-dated. If not effective-dated, this property returns a blank string.

This property is read-only.

TreeName

Description

This property returns the name of the tree as a string.

This property is read-only.

TreeSetId

Description

This property returns the SetID of the tree as a string if the tree has a SetID. If the tree doesn’t have a SetID, this property returns a blank string.

This property is read-only.

TreeUserKeyValue

Description

This property returns the UserKeyValue of the tree as a string if the tree has a UserKeyValue. If the tree doesn't have a UserKeyValue, this property returns a blank string.

This property is read-only.

Level Collection

Level collection is instantiated from a tree object with the Levels property.

See Chapter 33, Tree Classes , Levels, page 1731.

Level Collection Methods

In this section, we discuss the Level Collection methods. The methods are discussed in alphabetical order.

Add

Syntax

```
Add(LevelName)
```

Description

The **Add** method adds a *new* level called *LevelName* to the database. The specified level must be a *new* level, that is, it cannot exist in the database. *LevelName* takes a string value.

Note. *LevelName* must be 8 characters or less.

If no levels currently exist in the tree, the level is added at the top level. If levels already exist in the tree, the new level is added as the *last* level. You can change the level number of a level using the Number property.

If the new level is the first level, the **AllValuesAudit** property is automatically set to True.

The new level is *not*added to the database until the tree is explicitly saved.

This method returns a reference to the new level object.

See Also

Chapter 33, Tree Classes , Number, page 1682, Chapter 33, Tree Classes , AllValuesAudit, page 1682.

Item

Syntax

```
Item(LevelName, LevelNumber)
```

Description

The **Item** method returns a reference to the specified level in the level collection executing the method *as an object*. The *LevelName* parameter specifies the name of the level to access. This parameter takes a string value. The *LevelNumber* parameter specifies the number at which the level exists. This parameter takes a number value. For example, suppose your level collection contains the following levels, in this order:

1. CORPORATE
2. COMPANY
3. DIVISION
4. DEPARTMENT
5. BRANCH

You want to access the fourth level, DEPARTMENT. You would use the following code:

```
&MYLEVEL = &LVLCOLLECTION.Item("DEPARTMENT", 4);
```

Remove

Syntax

Remove()

Description

The **Remove** method deletes the current level *from the database*. You can use this method only after you have used the **First, Next,** or **Item** properties: otherwise the system doesn't know which level to delete from the tree. The rest of the levels in the collection after the deleted level are moved up in the list and renumbered so that the levels remain consecutively numbered.

Save

Syntax

Save()

Description

The **Save** method saves the current level to the database. You can use this method only after you have used the **First, Next,** or **Item** methods: otherwise the system doesn't know which level to save.

Note. If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Level Collection Properties

In this section, we discuss the Level collection properties. The properties are discussed in alphabetical order.

Count

Description

Returns the number of levels for the level collection object.

First

Description

The **First** property returns a reference to the first level in the level collection executing the method *as an open object*.

Last

Description

The **Last** property returns a reference to the last level in the level collection executing the method *as an open object*.

Next

Description

The **Next** property returns a reference to the next level in the level collection executing the method *as an open object*. You can use this method only after you have used either the **First** or **Item** properties: otherwise the system doesn't know where to start in the tree.

Level Class

Level objects are instantiated from the level class collection object with the Item method or one of the following properties:

- First
- Last
- Next

See Chapter 33, Tree Classes , Item, page 1678; Chapter 33, Tree Classes , First, page 1680; Chapter 33, Tree Classes , Last, page 1680 and Chapter 33, Tree Classes , Next, page 1680.

Level Class Methods

In this section, we discuss the Leaf class properties. The properties are discussed in alphabetical order.

Create

Syntax

Create(*LevelName, LevelNumber*)

Description

The **Create** method adds a *new* level called *LevelName*. The specified level must be a *new* level. You receive an error if the level already exists.

Parameters

Parameter	Description
LevelName	Specify the name of the new level. This parameter takes a string value.
LevelNumber	Specify the number of the new level. This parameter takes a number value.

Example

```
&LEVELCOLL = &MYTREE.Levels;
&LEVEL = &LEVELCOLL.First;
&LEVEL.Create("INTERNET", 9);
```

Delete

Syntax

Delete()

Description

The **Delete** method deletes the level object executing the method from the database.

Example

```
&MYLVLCOLL = &MYTREE.Levels;
&MYLVL = &MYLVLCOLL.Last;
&MYLVL.Delete();
```

Level Class Properties

In this section, we discuss the Level class properties. The properties are discussed in alphabetical order.

AllValuesAudit

Description

This property specifies whether PeopleSoft Tree Manager permits nodes to skip over this level. To allow nodes to skip this level, specify this parameter as True. If you are creating a new level, and this level is the first level in a tree, this property is automatically set to True. If the level isn't the first level, this property is automatically set to False.

This property is read-write.

Description

Description

This property returns the description of the level.

This property is read-write.

Name

Description

This property returns the name of the level.

This property is read-write.

Number

Description

This property returns the number of the level.

This property is read-write.

TreeBranchName

Description

This property returns the branch name of the tree as a string if the tree is branched. If not branched, this property returns a blank string.

This property is read-only.

TreeEffDt

Description

This property returns the effective date of the tree as a string if the tree is effective-dated. If not-effective dated, this property returns a blank string.

This property is read-only.

TreeName

Description

This property returns the name of the tree as a string.

This property is read-only.

TreeSetId

Description

This property returns the SetID of the tree as a string if the tree has a SetID. If the tree doesn't have a SetID, this property returns a blank string.

This property is read-only.

TreeUserKeyValue

Description

This property returns the UserKeyValue of the tree as a string if the tree has a UserKeyValue. If the tree doesn't have a UserKeyValue, this property returns a blank string.

Node Class

Node objects are instantiated from other tree classes, as follows:

- From a tree object with the FindNode method.
- From a leaf object with the Parent property.
- From another node object with the FirstChildNode, NextSib, PrevSib, and Parent properties.

See Chapter 33, Tree Classes , FindNode, page 1711; Chapter 33, Tree Classes , FirstChildNode, page 1696; Chapter 33, Tree Classes , NextSib, page 1699 and Chapter 33, Tree Classes , PrevSib, page 1700.

See Chapter 33, Tree Classes , Parent, page 1676.

See Chapter 33, Tree Classes , Parent, page 1700.

Node Class Methods

In this section, we discuss the Node class methods. The methods are discussed in alphabetical order.

Branch

Syntax

```
Branch()
```

Description

The **Branch** method branches the node executing this method, identifying all of its child nodes and leaves as part of that branch. *Branching* means taking a limb of a tree and creating another subtree to hold that limb. (Technically is it not creating a real tree.) The subtree is accessed through the Branches collection.

After you use the **Branch** method, you can no longer access the child nodes and leaves of the node that executed the method until you close the current tree, open the new branched tree, and find the node again. To unbranch the tree, use the **Unbranch** method.

Example

```
&MYNODE = &MYTREE.FindNode("10900", "");
&MYNODE.Branch();
```

Cut

Syntax

```
Cut()
```

Description

The **Cut** method cuts the node executing the method and puts it into a buffer (the clipboard). You can then use the PasteSib method to add the node back as a sibling of a different node. You can also use the PasteChild node method to add the node back as a child of a different node.

You can have only one object at a time on the clipboard. If you cut another leaf or node, the node on the clipboard is overwritten.

Parameters

None.

Returns

None.

See Also

Chapter 33, Tree Classes , PasteSib, page 1692 Chapter 33, Tree Classes , PasteChild, page 1692 Chapter 33, Tree Classes , Cut, page 1668 leaf method

Delete

Syntax

```
Delete()
```

Description

The **Delete** method deletes the node object executing the method *from the database.*

DeleteByName

Syntax

DeleteByName(*NodeName*)

Description

The **DeleteByName** method deletes the specified node *from the database*. The *NodeName* parameter takes a string value. The node specified by *NodeName* does *not* have to match the node executing the method. That is, if &MYNODE is associated with node 100200, you can specify a different node with the **DeleteByName** method. For example:

```
&MYNODE = &MYTREE.FindNode("100200", "");
/* some processing */
&MYNODE.DeleteByName("100300");
```

The node name specified by *NodeName* must be a valid node in the existing open tree object. If *NodeName* doesn't exist, you receive a runtime error.

Expand

Syntax

Expand(*ExpandType*)

Description

The **Expand** method gets the next level of nodes or leaves into memory from the selected node. What leaves or nodes get expanded depends on the *ExpandType*. Values are:

Value	Description
0	Expand only nodes
1	Expand nodes and leaves
2	Expand only one level

Example

```
If (&MYNODE.State = 2 AND &MYNODE.HasChildren);
   /* if node is collapsed */
   &MYNODE.Expand(2);
End-if;
```

InsertChildLeaf

Syntax

InsertChildLeaf(*RangeFrom*, *RangeTo*)

Description

The **InsertChildLeaf** method inserts a new leaf (as specified by the range parameters) under the node object executing the method.

A leaf object associated with the new leaf is returned.

The new leaf is inserted as the child of the current node, although there is no explicit ordering of leaves: leaves take the order of the alphanumeric database sort on their range fields.

Note. You must specify a range with this method. To insert a dynamic range leaf (that is, one with no range) use the **InsertDynChildLeaf** method.

Parameters

Parameter	Description
RangeFrom	Specify the starting range of the new leaf. This parameter takes a string value.
RangeTo	Specify the ending range of the new leaf. This parameter takes a string value.

Returns

A leaf object associated with the new leaf. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted is to test the result using either the All or None functions.

Example

```
&NEWLEAF = &MYLNODE.InsertChildLeaf("10090", "10100");

If None(&NEWLEAF) Then
   /* Error Processing */
End-If;
```

See Also

Chapter 33, Tree Classes , InsertDynChildLeaf, page 1688, Chapter 1, PeopleCode Built-in Functions, All , page 41, Chapter 1, PeopleCode Built-in Functions, None , page 335.

InsertChildNode

Syntax

InsertChildNode(*NodeName*)

Description

The **InsertChildNode** method inserts a new node (as specified by *NodeName*) as a child node under the node object executing the method. The node specified by *NodeName* must be a *new* node. You receive an error if the node already exists.

A node object associated with the new node is returned. If the new node isn't inserted successfully, the method returns False or Null.

Parameters

Parameter	Description
NodeName	Specify a name for the new node. This parameter takes a string value. *NodeName* must not already exist.

Returns

A reference to the new node. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted is to test the result using either the All or None functions.

Example

```
&NEWNODE = &MYNODE.InsertChildNode("100500");

If None(&NEWNODE) Then
   /* Do error processing */
End-If;
```

See Also

Chapter 1, PeopleCode Built-in Functions, All , page 41, Chapter 1, PeopleCode Built-in Functions, None , page 335.

InsertChildRecord

Syntax

InsertChildRecord(*NodeName*)

Description

The **InsertChildRecord** method inserts a new record (as specified by *NodeName*) as a node under the parent node executing this method. This method works only on trees that are Query Access Trees.

A node object associated with the new node is returned, otherwise this method returns False or Null.

Parameters

Parameter	Description
NodeName	Specify an existing record name. This parameter takes a string value. This record must not already be a node under the parent node (that is, a parent node can't have the same record as a child node more than once.)

Returns

A reference to the new node. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted is to test the result using either the All or None functions.

Example

```
&NEWNODE = &MYNODE.InsertChildRecord("PERSONAL_DATA");
```

```
If None(&NEWNODE) Then
   /* Do error processing */
End-if;
```

See Also

Chapter 1, PeopleCode Built-in Functions, All , page 41, Chapter 1, PeopleCode Built-in Functions, None , page 335.

InsertDynChildLeaf

Syntax

InsertDynChildLeaf()

Description

The **InsertDynChildLeaf** method inserts a dynamic leaf under the node object executing the method.

A leaf object associated with the new leaf is returned.

The new leaf is inserted as the child of the current node, although there is no explicit ordering of leaves.

Note. To insert a leaf with a specific range, use the **InsertChildLeaf** method.

Parameters

None.

Returns

A leaf object associated with the new leaf. If this method fails, it returns either False or a Null object reference, depending upon the type of error encountered. The best way to test whether the object was inserted is to test the result using either the All or None functions.

Example

```
&NEWLEAF = &MYLNODE.InsertDynChildLeaf();

If None(&NEWLEAF) Then
   /* Do error processing */
End-If;
```

See Also

Chapter 33, Tree Classes , InsertChildLeaf, page 1685, Chapter 1, PeopleCode Built-in Functions, All , page 41, Chapter 1, PeopleCode Built-in Functions, None , page 335.

InsertSib

Syntax

InsertSib(*NodeName*)

Description

The **InsertSib** method inserts a new node (as specified by *NodeName*) as a sibling node to the node object executing the method. The node specified by *NodeName* must be a *new* node. You receive an error if the node already exists.

A node object associated with the new node is returned. If the new node isn't inserted successfully, the method returns Null.

Parameters

Parameter	Description
NodeName	Specify a name for the new node. This parameter takes a string value. *NodeName* must not already exist.

Returns

A reference to the new node. If this method fails, it returns False.

Example

```
&NEWNODE = &MYNODE.InsertSib("100500");
```

InsertSibRecord

Syntax

InsertSibRecord(*NodeName*)

Description

The **InsertSibRecord** method inserts a new record (as specified by *NodeName*) as a sibling node of the parent node executing this method. This method works only on trees that are Query Access Trees.

A node object associated with the new node is returned, otherwise this method returns Null.

Parameters

Parameter	Description
NodeName	Specify an existing record name. This parameter takes a string value. This record must not already be a sibling node for the node executing the object (that is, a node can't have the same record as a node more than once.)

Returns

A reference to the new node. If this method fails, it returns False.

Example

```
&NEWNODE = &MYNODE.InsertSibRecord("PERSONAL_DATA");
```

MoveAsChild

Syntax

MoveAsChild(*Node*)

Description

The **MoveAsChild** method moves the node executing the method to a different node in the tree. The node becomes the first child node under the named node. The specified node becomes the new parent to the child node. The node specified by *Node* must be an existing node in the current tree. All child nodes and details are also moved with the node.

Parameters

Parameter	Description
Node	Specify a node object. This parameter value must be an object, not a string or a name.

Note. To specify a node name, not a node object, use the **MoveAsChildByName** method.

Example

The following example moves node 10200 from node 10100 (old parent) to node 00001 (new parent).

```
&MY_NODE = &MY_TREE.FindNode("10200", "");
&NEW_PARENT = &MY_TREE.FindNode("00001", "");
If &NEW_PARENT <> Null Then
   &MY_NODE.MoveAsChild(&NEW_PARENT);
End-If;
```

See Also

Chapter 33, Tree Classes , MoveAsChildByName, page 1690.

MoveAsChildByName

Syntax

MoveAsChildByName(*NodeName*)

Description

The **MoveAsChildByName** method moves the node executing the method to a different node in the tree. The node becomes the first child node under the parent node. The specified node becomes the new parent to the node. The node specified by *NodeName* must be a valid node name. All child nodes and details are also moved with the node.

Parameters

Parameter	Description
NodeName	Specify the name of a node. This parameter takes a string value. *NodeName* must be a valid node for the existing tree.

Note. To specify a node object, not a node name, use the **MoveAsChild** method.

Example

The following moves node 10200 from node 10100 (old parent) to node 00001 (new parent)

```
&MY_NODE = &MY_TREE.FindNode("10200", "");
If &MY_NODE <> Null Then
   &MY_NODE.MoveAsChildByName("00001");
End-If;
```

See Also

Chapter 33, Tree Classes , MoveAsChild, page 1690.

MoveAsSib

Syntax

MoveAsSib(*Node*)

Description

The **MoveAsSib** method moves the node executing the method to a new place in the tree. The current node becomes the next sibling node under the specified node. All child nodes and details are also moved with the node.

Parameters

Parameter	Description
Node	Specify a node object. This parameter value must be an object, not a string or a name.

Note. To specify a node name, not a node object, use the **MoveAsSibByName** method.

Example

```
&MYNODE = &MYTREE.FindNode("20000", "");
&MYNODE2 = &MYTREE.FindNode("20100", "");
&MYNODE.MoveAsSib(&MYNODE2);
```

See Also

Chapter 33, Tree Classes , MoveAsSibByName, page 1691.

MoveAsSibByName

Syntax

MoveAsSibByName(*NodeName*)

Description

The **MoveAsSibByName** method moves the node executing the method to a new place in the tree. The current node becomes the next sibling node under the specified node. All child nodes and details are also moved with the node.

Parameters

Parameter	Description
NodeName	Specify the name of a node. This parameter takes a string value. *NodeName* must be a valid node for the existing tree.

Note. To specify a node object, not a node name, use the **MoveAsSib** method.

Example

```
&MYNODE = &MYTREE.FindNode("20000", "");
&MYNODE.MoveAsSibByName("10100");
```

See Also

Chapter 33, Tree Classes , MoveAsSib, page 1691.

PasteChild

Syntax

PasteChild()

Description

The **PasteChild** method makes the node or leaf from the buffer (clipboard) a child to the node executing the method. You must use the **Cut** node or leaf method before you can paste a node or leaf.

You can have only one object at a time on the clipboard. If you cut another node or leaf, the node or leaf on the clipboard is overwritten.

Parameters

None.

Returns

None.

See Also

Chapter 33, Tree Classes , Cut, page 1668 Chapter 33, Tree Classes , PasteSib, page 1692
Chapter 33, Tree Classes , Cut, page 1684 node method

PasteSib

Syntax

PasteSib()

Description

The **PasteSib** method makes the node from the buffer (clipboard) a sibling to the node executing the method. You must use the **Cut** node method before you can paste a node.

You can have only one object at a time on the clipboard. If you cut another leaf or node, the node on the clipboard is overwritten.

Parameters

None.

Returns

None.

See Also

Chapter 33, Tree Classes , Cut, page 1668 Chapter 33, Tree Classes , PasteChild, page 1692 Chapter 33, Tree Classes , Cut, page 1684 node method

RefreshDescription

Syntax

RefreshDescription()

Description

The **RefreshDescription** method enables you to add a node or change the description of the node and have the update be displayed immediately. If you don't use RefreshDescription, you must save the tree and reopen it for the change to be displayed.

Parameters

None.

Returns

A zero (0) if description is refreshed successfully, a different error number otherwise.

Example

```
If &result = 1 Then
   GetTreeAndSessionObj(&Session, &TreeMgrObj);
   &NodeObj = &TreeMgrObj.findnode(&cSelectedNodeRec.GetField(Field.TREE_⇒
NODE).Value, "");
If All(&NodeObj) Then
   &NodeObj.refreshdescription();
End-If;
End-If;
```

Rename

Syntax

Rename(*NodeName*)

Description

The **Rename** method renames the node object executing the method without changing any other properties. The parameter *NodeName* takes a string value. The node specified by *NodeName* must be a *new* node. You receive an error if the node already exists. Also, the node specified by *NodeName* can't be branched.

Example

```
&MYNODE = &MYTREE.FindNode("10900", "");
&MYNODE.Rename("10200");
```

SwitchLevel

Syntax

SwitchLevel(*NewLevelNumber*)

Description

The **SwitchLevel** methods enables you to move a node down from one level to another.

NewLevelNumber must specify a valid level number. For trees in which the Level Use parameter is set to Strictly Enforce Levels, the new level must be at least one level *lower*than the node's parent level.

If the level specified by *NewLevelNumber* doesn't exist, it's automatically generated and added to the tree.

Parameters

Parameter	Description
NewLevelNumber	Specify the level number to where you want the node moved.

Returns

A zero (0) if node is moved successfully, a different error number otherwise.

Unbranch

Syntax

Unbranch()

Description

The **Unbranch** method is the opposite of the **Branch** method: that is, it unbranches the node executing the method if it is branched. All of the child node and leaves of the node become part of the current open tree and accessible in that tree. You can't unbranch the Root Node in a branched tree.

If the node executing the method isn't branched, you receive a runtime error. Use the **IsBranched** property to make sure a node is branched.

Example

```
&MYNODE = &MYTREE.FindNode("10900", "");
&MYNODE.Unbranch();
```

Node Class Properties

In this section, we discuss the Node class properties. The properties are discussed in alphabetical order.

AllChildCount

Description

This property returns the number of all the child nodes and leaves of the node, that is, all the nodes and leaves below this node. Values are 0 to 2 billion.

This property is read-only.

AllChildNodeCount

Description

This property returns the number of child nodes of the node, that is, all the nodes below this node.

This property is read-only.

Example

To determine all the leaves below a node, use the AllChildNodeCount with the AllChildCount property.

```
&AllCount = (&MyNode.AllChildCount - &MyNode.AllChildNodeCount);
```

ChildLeafCount

Description

This property returns the number of immediate child leaves of the node, that is, all leaves under this node.

This property is read-only.

ChildNodeCount

Description

This property returns the number of immediate child nodes of the node, that is, all nodes under this node.

This property is read-only.

Description

Description

This property returns the description of the node.

This property is read-only.

DisplayLevelNumber

Description

When using the Tree API to create a graphic representation of the tree (such as for HTML Tree Manager or other application pages) use this property to indicate the *display* level, that is, tell the user how many levels deep the node is. This is generally used for trees where levels aren't used and the LevelNumber property always returns 0.

This property is read-only.

See Also

Chapter 33, Tree Classes , LevelNumber, page 1699.

FirstChildLeaf

Description

This property returns a reference to the first child leaf of the node. This is the leaf that appears highest in the list of children of the node in PeopleSoft Tree Manager.

This property is read-only.

Example

```
&NEWLEAF = &MYNODE.FirstChildLeaf;
```

FirstChildNode

Description

This property returns a reference to the first child node of the node. This is the node that appears highest in the list of children of the node in PeopleSoft Tree Manager.

This property is read-only.

Example

```
&CHILDNODE = &MYNODE.FirstChildNode;
```

HasChildLeaves

Description

This property returns True if the node has immediate child leaves, False otherwise.

This property is read-only.

HasChildNodes

Description

This property returns True if the node has immediate child nodes, False otherwise.

This property is read-only.

HasChildren

Description

This property returns True if the node has *either* child leaves or child nodes anywhere in the subtree, not just immediately under the node.

This property is read-only.

HasNextSib

Description

This property returns True if the node has a next sibling, that is, if it isn't the last node.

This property is read-only.

Example

```
While &MYNODE.HasNextSib
   &MYNODE = &MYNODE.NextSib;
   /* do some processing */
End-While;
```

HasPrevSib

Description

This property returns True if the node has a previous sibling, that is, if it isn't the first node.

This property is read-only.

IsBranched

Description

This property returns True if the node is branched, False otherwise.

This property is read-only.

IsChanged

Description

This property returns True if the node has been edited or changed but the current tree hasn't been saved.

This property is read-only.

IsCut

Description

This property returns True if the node has been cut from the displayed tree. This property is generally used with the HTML Tree Manager.

This property is read-only.

IsDeleted

Description

This property returns True if the node has been deleted but the current tree hasn't been saved.

This property is read-only.

IsInserted

Description

This property returns True if the node was inserted as a new node into the tree but the tree hasn't been saved.

This property is read-only.

Example

```
If &MYNODE.IsInserted Then
   &MYTREE.Save();
End-if;
```

IsRoot

Description

This property returns True for both of the following:

- the node is the root node of an unbranched tree.
- the node is the top node of an opened tree branch.

Otherwise, the property returns False.

This property is read-only.

LastChildLeaf

Description

This property returns a reference to the last child leaf of the node. This is the leaf that appears lowest in the list of children of the node in PeopleSoft Tree Manager.

This property is read-only.

Example

```
&NEWLEAF = &MYNODE.LastChildLeaf;
```

LastChildNode

Description

This property returns a reference to the last child node of the node. This is the node that appears lowest in the list of children of the node in PeopleSoft Tree Manager.

This property is read-only.

Example

```
&CHILDNODE = &MYNODE.LastChildNode;
```

LevelNumber

Description

This property returns the level number of the node. Values are 1-99 for Strict or Loose level trees. This property returns 0 for trees that don't have levels.

This property is read-write.

Name

Description

This property returns the name of the node (as a string.) You must set this property to a valid value if you are creating a *new*node.

Note. Do not use this property to change the name of an existing node. The change will *not* be reflected in the database. You must use the **Rename** method to change the name of an existing node.

This property is read-write.

See Also

Chapter 33, Tree Classes , Rename, page 1693.

NextSib

Description

This property returns a reference to the next sibling node. If there isn't a next sibling node, Null is returned.

This property is read-only.

Example

```
While &MYNODE.HasNextSib
   &MYNODE = &MYNODE.NextSib;
   /* do some processing */
End-While;
```

Parent

Description

This property returns a reference to the node that is the parent of the node executing the property. If the current node has no parent (is a root node) Null is returned.

This property is read-only.

Example

```
&PARENT = &MYNODE.Parent;
```

PrevSib

Description

This property returns a reference to the node that is the previous sibling node to the node executing the property. If there isn't a previous sibling, Null is returned.

This property is read-only.

Example

```
If &MYNODE.PrevSib Then
   /* do processing */
End-if;
```

State

Description

This property returns the *state* of the node, that is, does it have children, and are they expanded or collapsed.

See **Expanded** method.

The values for this property are:

Value	Description
0	Node has no children
1	Children are expanded
2	Children are collapsed
3	Leaves are collapsed

This property is read-only.

Example

```
&VALUE = &MYNODE.State;
If &VALUE = 2 Then
   &MYNODE.Expand(0);
End-if;
```

TreeBranchName

Description

This property returns the branch name of the tree as a string if the tree is branched. If not branched, this property returns a blank string.

This property is read-only.

TreeEffDt

Description

This property returns the effective date of the tree as a string if the tree is effective-dated. If not effective-dated, this property returns a blank string.

This property is read-only.

TreeName

Description

This property returns the name of the tree as a string.

This property is read-only.

TreeSetId

Description

This property returns the SetID of the tree as a string if the tree has a SetID. If the tree doesn’t have a SetID, this property returns a blank string.

This property is read-only.

TreeUserKeyValue

Description

This property returns the UserKeyValue of the tree as a string if the tree has a UserKeyValue. If the tree doesn’t have a UserKeyValue, this property returns a blank string.

Type

Description

This property returns the type of the node. This property takes a string value. Values are:

- "G" (Group): normal unbranched node or record group
- "B" (Branched nodes)
- "R" (Query Record): used for Query Access Trees only

This property is read-only.

Tree Class

Tree objects are instantiated from the following:

- From a session object with the GetTree method.
- From a branch collection with the First, Last, or Next properties or the Item method.
- From a tree collection with the First, Last, or Next properties or the Item method.

The following code sample gets a tree, then opens the tree structure associated with that tree:

```
Local string &TREE_NAME;
Local string &TREE_DT;
Local ApiObject &MYSESSION, &VC_TREE, &STRUCT;


/* Get and Open the Tree using the Tree API */
&MYSESSION = %Session;

/* Get the Tree Name and Effective Date from the level 0 record */
&TREE_NAME = "CUSTOMER";
&TREE_DT = "1900-01-01";

/* Get and Open the Tree  */

&VC_TREE = &MYSESSION.GetTree("", "", "", "", "");
&VC_TREE.OPEN("BKINV", "", &TREE_NAME, &TREE_DT, "", False);

/* Get and Open Tree Structure */

&STRUCT = VC_TREE.GetStructure;
```

See Chapter 33, Tree Classes , GetTree, page 1665; Chapter 33, Tree Classes , First, page 1667; Chapter 33, Tree Classes , Last, page 1668; Chapter 33, Tree Classes , Next, page 1668 and Chapter 33, Tree Classes , Item, page 1666.

Tree Class Methods

In this section, we discuss the Tree class methods. The methods are discussed in alphabetical order.

Audit

Syntax

```
Audit()
```

Description

The **Audit** method audits the tree object executing the method to determine its validity.

The **Audit** method can be used only on an open tree, not on a closed tree. This means you must have opened the tree with the Open method before you can audit it.

This method returns a Boolean value: True if the tree passes all audits, False if one or more errors occurred. If **Audit** returns False, an error is logged.

See Also

Chapter 33, Tree Classes , Open, page 1715, .

AuditByName

Syntax

```
AuditByName(SetID, UserKeyValue, TreeName, EffDt, BranchName)
```

Description

The **AuditByName** method audits the tree specified by the parameters passed to it. The AuditByName method can be used only with a tree identifier, it cannot be used on a fully instantiated, open tree. Before you use the AuditByName method, you must explicitly close any open tree objects (with the Close method.) You receive an error if there are any open trees.

Parameters

Parameter	Description
SetID	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
UserKeyValue	Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
TreeName	Specify the name for this tree. This parameter takes a string value.
EffDt	Specify the effective date for this tree. This parameter takes a string value.
BranchName	This parameter is required, but it is unused in this release. You must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

This method returns a Boolean value: True if the tree passes all audits, False if one or more errors occurred. If **AuditByName** returns False, a runtime error occurs, and either a message box or a logged error reports the nature of the error found in the tree.

Example

```
&ISVALID = &MYTREE.AuditByName("", "", "PERSONAL_DATA", "05-05-1997", "");
```

See Also

Chapter 33, Tree Classes , Close, page 1704, Chapter 33, Tree Classes , IndirectionMethod, page 1745, .

Close

Syntax

```
Close()
```

Description

The **Close** method closes the tree, freeing the memory associated with that object, and discarding any changes made to the tree since the last save. The Close method can be used only on an open tree, not a closed tree. This means you must have opened the tree with the Open method before you can close it. To save any changes, you must use the **Save** method before using Close.

You must close your tree when you finished processing. Canceling out of a page does *not* close a tree. You may receive error messages every other time you run your program if you haven't closed your trees.

See Also

Chapter 33, Tree Classes , Open, page 1715.

Copy

Syntax

Copy(*FromSetId, FromUserKeyValue, FromTreeName, FromEffDt, FromBranchName, ToSetId, ToUserKeyValue, ToTreeName, ToEffDt, ToBranchName*)

Description

The **Copy** method copies from the tree identified by the *From* parameters and copies it to the tree identified with the *To* parameters. The tree specified by the *To* parameters must be a *new* tree. You receive an error if the tree already exists. The tree specified by the *From* parameters does *not* have to match the tree identifier executing the method.

The **Copy** method can be used only with a *closed* tree, it cannot be used on an open tree. Before you use the **Copy** method, you must explicitly close any open tree objects (with the Close method.) You receive an error if there are any open trees.

To access the new tree, you must use the Open method.

Parameters

Parameter	Description
FromSetId	Specify the table indirection key for the tree to be copied from. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
FromUserKeyValue	Specify the User Key Value for the tree to be copied from. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
FromTreeName	Specify the name for the tree to be copied from. This parameter takes a string value.
FromEffDt	Specify the effective date for the tree to be copied from. This parameter takes a string value.
FromBranchName	This parameter is required, but it is unused in this release. You must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
ToSetId	Specify the table indirection key for the tree to be copied to. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
ToUserKeyValue	Specify the User Key Value for the tree to be copied to. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B" you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
ToTreeName	Specify the name for the tree to be copied to. This parameter takes a string value.
ToEffDt	Specify the effective date for the tree to be copied to. This parameter takes a string value.
ToBranchName	This parameter is required, but it is unused in this release. You must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

An integer: 0 if copied successfully.

Example

```
&MYTREE.Copy("","","PERSONAL_DATA", "05-05-1997", "", "","","PERSONAL_EDIT1", ⇒
"05-05-1997", "");
```

See Also

Chapter 33, Tree Classes , Close, page 1704, Chapter 33, Tree Classes , Open, page 1715, Chapter 33, Tree Classes , IndirectionMethod, page 1745.

Create

Syntax

Create(*SetID, UserKeyValue, TreeName, EffDt, StructureName*)

Description

The **Create** method creates a new tree, based on the parameters passed with the method. The specified tree must be a *new* tree. You receive an error if the tree already exists.

The Create method can be used only with a *closed* tree, it cannot be used on an open tree. Before you use the Create method, you must explicitly close any open tree objects (with the **Close** method.) You receive an error if there are any open trees.

After you create a new tree, you don't have to open it with the Open method. The existing tree object points to the new tree.

The tree structure specified with *StructureName* determines what database fields the new tree is based on, what pages you can use to create tree nodes and detail values, and what record definitions PeopleSoft Tree Manager saves tree-related data in. The tree structure must already exist.

The new tree must be saved (**Save, SaveAs,** and so on) for this tree to be saved to the database.

Parameters

Parameter	Description
SetID	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
UserKeyValue	Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
TreeName	Specify the name for this tree. This parameter takes a string value.
EffDt	Specify the effective date for this tree. This parameter takes a string value.
StructureID	Specify the name of the tree structure to use for this tree. This parameter takes a string value.

Returns

An integer: 0 if created successfully.

Example

```
&MYTREE.Create("","","PERSONAL_NEW", "05-05-1997", "PERSONAL_DATA");
```

See Also

Chapter 33, Tree Classes , Close, page 1704, Chapter 33, Tree Classes , IndirectionMethod, page 1745, .

Delete

Syntax

Delete(*SetID, UserKeyValue, TreeName, EffDt, BranchName*)

Description

The **Delete** method deletes the specified tree *from the database*. The **Delete** method can be used only with a *closed* tree, it cannot be used on an open tree. Before you use the **Delete** method, you must explicitly close any open tree objects (with the **Close** method.) You receive an error if there are any open trees.

The specified tree does *not* have to match the tree identifier executing the method.

Parameters

Parameter	Description
SetID	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
UserKeyValue	Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
TreeName	Specify the name for this tree. This parameter takes a string value.
EffDt	Specify the effective date for this tree. This parameter takes a string value.
BranchName	Specify the name of the branch for the tree. This parameter takes a string value. If the tree is unbranched, you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

An integer: 0 if deleted successfully.

Example

```
&MYTREE.Delete("","","PERSONAL_REN2", "05-05-1997", "");
```

See Also

Chapter 33, Tree Classes , Close, page 1704, Chapter 33, Tree Classes , IndirectionMethod, page 1745.

Exists

Syntax

Exists(*SetID*, *UserKeyValue*, *TreeName*, *EffDt*, *BranchName*)

Description

The **Exists** method finds an existing tree, as specified by the parameters. The parameters must specify a unique tree. If a tree matching the parameters is found, the method returns 0.

Parameters

Parameter	Description
SetID	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
UserKeyValue	Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
TreeName	Specify the name for this tree. This parameter takes a string value.
EffDt	Specify the effective date for this tree. This parameter takes a string value.
BranchName	This parameter is required, but it is unused in this release. You must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

An integer: 0 if tree exists.

Example

```
&RSLT = &MYTREE.Exists("", "", "VENDOR_PER_DATA", "05-03-1998", "");

If &RSLT = 0 Then
   /* tree exists, do processing */
Else
   /* tree doesn't exist, do error processing */
End-if;
```

See Also

Chapter 33, Tree Classes , IndirectionMethod, page 1745.

FindLeaf

Syntax

FindLeaf(*RangeFrom, RangeTo*)

Description

The **FindLeaf** method finds an existing leaf (specified by the *range* parameters) in the tree executing this method. If the leaf is found, a leaf object is returned. If the leaf isn't found, a Null is returned.

You can use wildcards with the *range* parameters. Use an asterisk (*) in place of a number of characters. Use a question mark (?) in place of one character. You can also provide one of the ranges, and specify a Null string for the other.

Suppose you wanted to find the leaf with the range 81005 to 82000. The following would be valid:

```
&MyLeaf = &MyTree.FindLeaf("81*", "82000");
```

This would also be valid:

```
&MyLeaf = &MyTree.FindLeaf("8100?", "82000");
```

You could also specify this:

```
&MYLEAF = &MYTREE.FindLeaf("81000", "");
```

If you specify the exact same value for both *RangeFrom* and *RangeTo*, the *RangeTo* parameter is ignored.

Parameters

Parameter	Description
RangeFrom	Specify the starting range of the leaf to be found. This parameter takes a string value.
RangeTo	Specify the ending range of the leaf to be found. This parameter takes a string value.

Returns

A reference to a leaf object.

Example

```
&MYLEAF = &MYTREE.FindLeaf("81000", "");
```

FindNextDisplayObject

Syntax

FindNextDisplayObject(*&Node, &Leaf*)

Description

The **FindNextDisplayObject** method sets either the **DisplayLeaf** or **DisplayNode** properties with the next displayed leaf or node, following the specified node or leaf. This property is primarily used with trees displayed in a graphical nature, such as with the HTML tree. If the next display object isn't found (that is, you're at the bottom of the tree), both properties are set to Null.

Note. If the node or leaf you're searching for is in an unexpanded portion of the tree, this method expands the tree down to that node or leaf.

You should specify *either* a node *or* a leaf, *not* both. You should specify a Null string for the other. If both are provided, only the node is used and the leaf is ignored.

Parameters

Parameter	Description
&Node	Specify an already existing node object. Specify either a node or a leaf.
&Leaf	Specify an already existing leaf object. Specify either a node or a leaf.

Returns

None. Sets either DisplayLeaf or DisplayNode properties.

Example

```
&TreeObj.FindNextDisplayObject(&NodeObj,Null);

If None(&TreeObj.DisplayNode) Then    /* Next object isn't a node */
   If All(&TreeObj.DisplayLeaf) Then      /* Next object is a Leaf */
      &NextObj = &TreeObj.DisplayLeaf;
   Else
      /* code for being at end of tree */      /* no next object   */
   End-if;
Else
   &NextObj = &TreeObj.DisplayNode;
End-if;
```

See Also

Chapter 33, Tree Classes , DisplayLeaf, page 1725, Chapter 33, Tree Classes , DisplayNode, page 1726, Chapter 33, Tree Classes , FindPrevDisplayObject, page 1712.

FindNode

Syntax

FindNode(*NodeName, Description*)

Description

The **FindNode** method returns a reference to the node object specified by the parameters passed to the method. If the node is found, a node object is returned. If the node isn't found, a Null is returned.

Note. If the node you're searching for is in an unexpanded portion of the tree, this method expands the tree down to that node.

You should specify *either*the node name *or*the description, *not*both. You should specify a Null string for the other. If both are provided, only the node name is used and the description is ignored.

You can use wildcards with the parameters. Use an asterisk (*) in place of a number of characters. Use a question mark (?) in place of one character.

Suppose you wanted to find the node called 10200, with a description of "Human Resources". The following would be valid:

```
&MyNode = &MyTree.FindNode("", "Human*");
```

This would also be valid:

```
&MyNode = &MyTree.FindNode("1020?", "");
```

You could also use the following:

```
&MYNODE = &MYTREE.FindNode("10200", "");
```

Parameters

Parameter	Description
NodeName	Specify the name of the node that you want to find. This parameter takes a string value. Specify either the node name or the description.
Description	Specify the description of the node that you want to find. This parameter takes a string value. Specify either the node name or the description.

Returns

A reference to a node object.

Example

```
&MYNODE = &MYTREE.FindNode("10200", "");
```

FindPrevDisplayObject

Syntax

FindPrevDisplayObject(*&Node, &Leaf*)

Description

The **FindPrevDisplayObject** method sets either the DisplayLeaf or DisplayNode properties with the previous displayed leaf or node, following the specified node or leaf. This property is primarily used with trees displayed in a graphical nature, such as with the HTML tree. If the previous display object isn't found (that is, you're at the top of the tree), both properties are set to Null.

Note. If the node or leaf you're searching for is in an unexpanded portion of the tree, this method expands the tree down to that node or leaf.

You should specify *either* a node *or* a leaf, *not* both. You should specify a Null string for the other. If both are provided, only the node is used and the leaf is ignored.

Parameters

Parameter	Description
&Node	Specify an already existing node object. Specify either a node or a leaf.
&Leaf	Specify an already existing leaf object. Specify either a node or a leaf.

Returns

None. Sets either DisplayLeaf or DisplayNode properties.

Example

```
&TreeObj.FindPrevDisplayObject(&NodeObj,Null);

If None(&TreeObj.DisplayNode) Then    /* Prev object isn't a node */
   If All(&TreeObj.DisplayLeaf) Then      /* Prev object is a Leaf */
      &PrevObj = &TreeObj.DisplayLeaf;
   Else
      /* code for being at beginning of tree */      /* no Prev object   */
   End-if;
Else
   &PrevObj = &TreeObj.DisplayNode;
End-if;
```

See Also

Chapter 33, Tree Classes , DisplayLeaf, page 1725, Chapter 33, Tree Classes , DisplayNode, page 1726, Chapter 33, Tree Classes , FindNextDisplayObject, page 1710.

FindRoot

Syntax

FindRoot()

Description

The **FindRoot** method returns a reference to the root node object of the tree object executing the method. This method can be used only on an open tree, not on a closed tree. This means you must have opened the tree with the Open method before you can find the root node.

This method returns an error if the tree has no nodes associated with it.

Returns

A reference to a node object.

Example

```
&MYNODE = &MYTREE.FindRoot();
```

See Also

Chapter 33, Tree Classes , Open, page 1715.

InsertRoot

Syntax

InsertRoot(*NodeName*)

Description

The **InsertRoot** method inserts a node at the top of the tree object executing the method. *NodeName* is the name of the node you are creating. This parameter takes a string value. The **InsertRoot** method can be used only on an open tree, not on a closed tree. This means you must have opened the tree with the Open method before you can insert any nodes.

You can insert the root node only in a new tree, that is, a tree that was just created with the **Create** method. Also, after you create a new tree, you must insert the root node before you can insert any other nodes.

In a strict level tree, the first level must be defined before you can insert the root node.

This method returns an error if the tree already has nodes in it.

Example

```
&MYTREE.InsertRoot("000001");
```

See Also

Chapter 33, Tree Classes , Open, page 1715.

LeafExists

Syntax

LeafExists(*RangeFrom, RangeTo*)

Description

The **LeafExists** method enables you to verify if the leaf specified by the parameters exists in the current tree. Use this method before using FindLeaf to verify a leaf exists before searching for it.

Note. This method searches only in the expanded portions of a tree. If a node isn't expanded, this method won't search that portion of the tree.

You can use wildcards with the *range* parameters. Use an asterisk (*) in place of a number of characters. Use a question mark (?) in place of one character. You can also provide one of the ranges, and specify a Null string for the other.

Suppose you wanted to check the leaf with the range 81005 to 82000 exists. The following would be valid:

```
&MyResult = &MyTree.LeafExists("81*", "82000");
```

This would also be valid:

```
&MyResult = &MyTree.LeafExists("8100?", "82000");
```

You could also specify this:

```
&MyResult = &MYTREE.LeafExists("81000", "");
```

If you specify the exact same value for both *RangeFrom* and *RangeTo*, the *RangeTo* parameter is ignored.

Parameters

Parameter	Description
RangeFrom	Specify the starting range of the leaf you're verifying. This parameter takes a string value.
RangeTo	Specify the ending range of the leaf you're verifying. This parameter takes a string value.

Returns

An integer: 0 if the leaf exists.

See Also

Chapter 33, Tree Classes , NodeExists, page 1718, Chapter 33, Tree Classes , FindLeaf, page 1710.

Open

Syntax

```
Open(SetID, UserKeyValue, TreeName, EffDt, BranchName, Update)
```

Description

The **Open** method opens the tree object specified by the parameters. The **Open** method can be used only with a *closed* tree, it cannot be used on an open tree. You cannot read or set any properties of a tree until after you open it. When you open a tree, you can specify whether you want to open it for update, or in read-only mode, with the *Update* parameter. If you try updating or writing to a tree opened in read-only mode, you receive an error.

Parameters

Parameter	Description
SetID	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
UserKeyValue	Specify the User Key value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
TreeName	Specify the name for the tree. This parameter takes a string value.
EffDt	Specify the effective date for the tree. This parameter takes a string value. The format must be in the correct format for the database platform the code is executing on.
BranchName	This parameter is required, but it is unused in this release. You must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
Update	Specify if you want to open the tree for update or in read-only mode. This parameter takes a Boolean value. Values for *Update* are: • True: open tree structure in update mode (read-write). • False: open tree structure in read-only mode. If you try updating or writing to a tree structure opened in read-only mode, you receive an error.

Returns

An integer: 0 if opened successfully.

Example

The following example opens a tree object, then tests to see if the tree was actually opened:

```
&RSLT = &MYTREE.Open("","","PERSONAL_DATA", "05-05-1997", "", True);

If &RSLT = 0 Then
   /* normal processing */
Else
   /* error processing tree isn't open */
End-if;
```

See Also

Chapter 33, Tree Classes , Close, page 1704, Chapter 33, Tree Classes , IndirectionMethod, page 1745.

OpenWholeTree

Syntax

OpenWholeTree(*SetID*, *UserKeyValue*, *TreeName*, *EffDt*)

Description

The **OpenWholeTree** method enables you to open a tree as if it were an unbranched tree. With this kind of tree, branched nodes *cannot* be distinguished from unbranched nodes. This means you can traverse the entire tree. It also means that branches have a node icon, not a branch icon. This method can be used only with a *closed* tree, it cannot be used on an open tree. You cannot read any properties of a tree until after you open it. This tree is always opened as read-only.

Note. You can't open a tree using this method and update it. The tree is opened in read-only mode.

Parameters

Parameter	Description
SetID	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
UserKeyValue	Specify the User Key value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
TreeName	Specify the name for the tree. This parameter takes a string value.
EffDt	Specify the effective date for the tree. This parameter takes a string value. The format must be in the correct format for the database platform the code is executing on.

Returns

An integer: 0 if opened successfully.

Example

```
&pSession = %Session;

&pTreeObj = &pSession.GetTree();
&status = &pTreeObj.OpenWholeTree(&Setid, &SetCntrlValue, &TreeName, ⇒
String(&EffDt));
```

See Also

Chapter 33, Tree Classes , Close, page 1704, Chapter 33, Tree Classes , IndirectionMethod, page 1745.

NodeExists

Syntax

NodeExists(*NodeName*)

Description

The **NodeExists** method enables you to verify if the node specified by *NodeName* exists in the current tree. Use this method before using FindNode to verify a node exists before searching for it.

Note. If the node you're searching for is in an unexpanded portion of the tree, this method expands the tree down to that node.

Parameters

Parameter	Description
NodeName	Specify the name of the node that you want to find. This parameter takes a string value.

Returns

An integer: 0 if the node exists.

See Also

Chapter 33, Tree Classes , FindLeaf, page 1710, Chapter 33, Tree Classes , FindNode, page 1711, Chapter 33, Tree Classes , LeafExists, page 1714.

Rename

Syntax

Rename(*FromSetId, FromUserKeyValue, FromTreeName, FromEffDt, FromBranchName, ToTreeName*)

Description

The **Rename** method renames a tree specified with the *From* parameters to the tree specified with *ToTreeName*. The tree specified by *ToTreeName* must be a *new* tree. You receive an error if the tree already exists. The tree specified by the *From* parameters does *not* have to match the tree executing the method.

The **Rename** method can be used only with a *closed* tree, it cannot be used on an open tree. Before you use the **Rename** method, you must explicitly close any open tree objects (with the Close method.) You receive an error if there are any open trees.

To access the new tree, you must use the Open method.

Parameters

Parameter	Description
FromSetId	Specify the table indirection key for the tree to be copied from. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
FromUserKeyValue	Specify the User Key Value for the tree to be copied from. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B" you must enter a Null string (that is, two quotation marks with no blank space between them (")) for this parameter.
FromTreeName	Specify the name for the tree to be copied from. This parameter takes a string value.
FromEffDt	Specify the effective date for the tree to be copied from. This parameter takes a string value.
FromBranchName	This parameter is required, but it is unused in this release. You must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
ToTreeName	Specify the new name of the tree. This parameter takes a string value.

Returns

An integer: 0 if renamed successfully.

Example

```
&MYTREE.Rename("","","PERSONAL_DATA3", "05-05-1997","", "PERSONAL_REN2");
```

See Also

Chapter 33, Tree Classes , Close, page 1704, Chapter 33, Tree Classes , Open, page 1715, Chapter 33, Tree Classes , Close, page 1704, Chapter 33, Tree Classes , IndirectionMethod, page 1745.

Save

Syntax

Save()

Description

The **Save** method writes any changes to the tree executing the method to the database. It also performs audits.

The **Save** method can be used only on an open tree, not on a closed tree. This means you must have opened the tree with the **Open** method before you can save it.

The tree object remains open after executing **Save**. You must execute the **Close** method on the object before it is closed and the memory freed.

This method returns an optional Boolean value: True if the tree was saved with no errors, False if there are one or more errors.

Note. If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Parameters

None.

Returns

An integer: 0 if save successfully.

SaveAs

Syntax

SaveAs(*SetID, UserKeyValue, TreeName, EffDt, BranchName*)

Description

The **SaveAs** method writes any changes to the tree executing the method to the new tree specified with the parameters. It also performs audits.

The tree specified with the parameters must be a *new* tree. You receive an error if the tree already exists. The new tree must have at least one different key field from the original tree.

The **SaveAs** method automatically closes the tree that is associated with the tree object executing the method, *and* associates that tree object with the new tree. Any changes made to the original tree since the last time the tree was saved are discarded. For example, suppose you opened a tree named DEPENDENT_BENEF and associated it with the variable &MYTREE. If you executed the SaveAs method with &MYTREE, changing the name to DEPENDENT_BENEF2, &MYTREE would now be associated with DEPENDENT_BENEF2, and the original tree, DEPENDENT_BENEF, would be closed.

```
&MYTREE.Open("", "", "DEPENDENT_BENEF", "02-02-1999", "", False);
/* do some processing */
&MYTREE.SaveAs("", "", "DEPENDENT_BENEF2", "02-05-1999", "");
```

&MYTREE is now associated with DEPENDENT_BENEF2, not with the original open tree.

Note. If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Parameters

Parameter	Description
SetID	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them (("")) for this parameter.
UserKeyValue	Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them (("")) for this parameter.
TreeName	Specify the name for this tree. This parameter takes a string value.
EffDt	Specify the effective date for this tree. This parameter takes a string value.
BranchName	This parameter is required, but it is unused in this release. You must enter a Null string (that is, two quotation marks with no blank space between them (("")) for this parameter.

Returns

This method returns an integer: 0 if saved successfully, a number greater than 0 otherwise.

Example

```
&MYTREE.SaveAs("", "", "DEPENDENT_BENEF2", "02-05-1999", "");
```

SaveAsDraft

Syntax

SaveAsDraft(*SetID, UserKeyValue, TreeName, EffDt, BranchName*)

Description

The **SaveAsDraft** method writes any changes to the tree executing the method to the new tree specified with the parameters. However, it does *not* perform audits. Trees that are saved in this way are marked as "invalid" and will not be valid (that is, cannot be used with other PeopleTools) until the tree audit is passed during Save or SaveAs.

The tree specified with the parameters must be a *new* tree. You receive an error if the tree already exists. The new tree must have at least one different key field from the original tree.

The **SaveAsDraft** method automatically closes the tree that is associated with the tree object executing the method, *and* associates that tree object with the new tree. Any changes made to the original tree since the last time the tree was saved are discarded.

See SaveAs for more details.

Note. If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Parameters

Parameter	Description
SetID	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
UserKeyValue	Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
TreeName	Specify the name for this tree. This parameter takes a string value.
EffDt	Specify the effective date for this tree. This parameter takes a string value.
BranchName	This parameter is required, but it is unused in this release. You must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Returns

This method returns an integer: 0 if saved successfully, a number greater than 0 otherwise.

Example

```
&MYTREE.  SaveAsDraft("","","PERSONAL_EDIT2", "05-05-1997", "");
```

SaveDraft

Syntax

SaveDraft()

Description

The **SaveDraft** method writes any changes to the tree executing the method to the database. However, it does *not* perform audits. Trees that are saved in this way are marked as "invalid" and will not be valid (that is, cannot be used with other PeopleTools) until the tree audit is passed.

The **SaveDraft** method can be used only on an open tree, not on a closed tree. This means you must have opened the tree with the Open method before you can save it.

The tree object remains open after executing **SaveDraft**. You must execute the Close method on the object before it is closed and the memory freed.

Note. If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Parameters

None.

Returns

This method returns an integer: 0 if saved successfully, a number greater than 0 otherwise.

See Also

Chapter 33, Tree Classes , Close, page 1704, Chapter 33, Tree Classes , Open, page 1715, .

Tree Class Properties

In this section, we discuss the Tree class properties. The properties are discussed in alphabetical order.

AllValues

Description

When the tree is audited, the **AllValues** property specifies whether the audit should verify that the tree includes all user data detail values. This property returns a Boolean value: True, check all values, False, do not check all values.

If this is a new tree, and you do not set this property, a default value of False is automatically set.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

```
&MyTree.AllValues = False;
```

AuditDetails

Description

Generally, when you create a tree structure with Detail records or fields in your tree structure, the terminal nodes of the trees based on this structure have details. However, in certain cases (for example, in the PeopleSoft Internet Architecture) the trees based on this structure of your terminal nodes won't have details. In these cases, set this property to False.

The default value for this property is True.

Branches

Description

This property returns a reference to a branch collection object. This property is used only with branched trees: it returns Null for trees that aren't branched.

See Branch Collection Class Methods.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

BranchLevel

Description

This property returns the level number (as a number) where this branch occurs in the tree. This property is used only with trees with levels: it returns zero for trees that don't have levels.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Example

```
&LevelNumber = &MyTree.BranchLevel;
```

BranchName

Description

This property returns the name of the specific branch for the tree, as a string. This property is used only with branched trees: it returns Null for trees that aren't branched.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Example

```
&BranchName = &MyTree.BranchName;
```

Category

Description

This property returns the category name for the tree, as a string. The category is a high-level grouping under which you can organize your tree structures and tree definitions. If you are creating a new tree, this property is required.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

```
&MyTree.Category = "PurchaseOrders";
```

CheckLeafUserData

Description

In PeopleSoft Tree Manager, the details must already exist when they are inserted into the tree. However, sometimes this is impractical with the Tree API. If you're inserting nodes and leaves to create a new tree, and the details don't exist, set this property to False.

The default value for this property is True.

This property is read-write.

Description

Description

This property returns the description of the tree, as a string. If you are creating a new tree, this property is required.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

```
&MyTree.Description = "Departmental Security";
```

DisplayLeaf

Description

This property specifies whether a leaf is displayed. This property is set by the FindNextDisplayObject or FindPreviousDisplayObject methods.

This property is read-only.

See Also

Chapter 33, Tree Classes , FindNextDisplayObject, page 1710, Chapter 33, Tree Classes , FindPrevDisplayObject, page 1712.

DisplayNode

Description

This property specifies whether a node is displayed. This property is set by the FindNextDisplayObject or FindPreviousDisplayObject methods.

This property is read-only.

See Also

Chapter 33, Tree Classes , FindNextDisplayObject, page 1710, Chapter 33, Tree Classes , FindPrevDisplayObject, page 1712.

DuplicateLeaves

Description

In most trees, you want each detail value to appear only once in the tree. To permit duplicate values, set this property to True.

If this is a new tree, and you do not set this property, a default value of False is automatically set.

This property can be only used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

EffDt

Description

This property returns the effective date of the tree, as a date.

If this is a new tree, and you do not set this property, a default value of the current date is automatically set.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

The following example uses a PeopleCode system variable for setting the effective date.

```
&MYTREE.EffDt = %ClientDate;
```

HasDetailRanges

Description

This property returns True if the tree has any detail ranges.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

IsBranched

Description

This property indicates whether the tree is branched. Possible values for this property are True and False: True if the tree is branched, False if it isn't.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Example

```
If &MyTree.IsBranched Then
   /* Do branched processing */
Else
   /*Do unbranched processing */
End-If;
```

IsChanged

Description

This property returns True if the tree has been changed but not saved, False if the tree is unchanged.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Example

```
If &MyTree.IsChanged Then
   &MyTree.Save();
End-if;
&MyTree.Close();
```

IsOpen

Description

This property returns True if the tree is open, False if the tree is closed.

This property is read-only.

Example

```
If Not &MyTree.IsOpen Then
   &MyTree.Open("","","PERSONAL_DATA", &Date, "", True);
End-If;
```

IsQueryTree

Description

This property returns True if the tree is a Query access tree. It returns False if the tree is not a Query Access Tree or a regular branched or unbranched tree.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

IsReadOnly

Description

This parameter indicates whether the tree can be updated.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Example

```
If &MyTree.IsReadOnly Then
   /* issue errors - can only process updateable tree */
Else
   /* do processing */
End-If;
```

IsValid

Description

This property returns True if the tree is a valid tree that has passed all audits. It returns False otherwise.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

IsWholeTree

Description

This parameter indicates whether the entire tree is accessible. For example, if you open a tree that is branched, and don’t have access to one or more of the branches, this parameter returns False.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

Example

```
If &MyTree.IsWholeTree Then
   /* Do processing */
Else
   /* close tree and reopen using OpenWholeTree method */
End-If;
```

KeyBranchName

Description

This property returns the branch name of the tree as a string if the tree is branched. If not branched, this property returns a blank string.

This property can be used with a closed tree, that is, before you use the **Open** method. You can use this property to open a tree without explicitly knowing its branch

This property is read-only.

KeyEffDt

Description

This property returns the effective date of the tree as a string if the tree is effective-dated. If not effective-dated, this property returns a blank string.

This property can be used with a closed tree, that is, before you use the **Open** method. You can use this property to open a tree without explicitly knowing its effective date.

This property is read-only.

KeyName

Description

This property returns the name of the tree as a string.

This property can be used with a closed tree, that is, before you use the **Open** method. You can use this property to open a tree without explicitly knowing its name.

If the tree has already been opened, you can use the **Name** property to get the tree name, or to create a new tree.

This property is read-only.

KeySetId

Description

This property returns the SetID of the tree as a string if the tree has a SetID. If the tree doesn’t have a SetID, this property returns a blank string.

This property can be used with a closed tree, that is, before you use the **Open** method. You can use this property to open a tree without explicitly knowing its SetID.

This property is read-only.

KeyUserKeyValue

Description

This property returns the UserKeyValue of the tree as a string if the tree has a UserKeyValue. If the tree doesn't have a UserKeyValue, this property returns a blank string.

This property can be used with a closed tree, that is, before you use the **Open** method. You can use this property to open a tree without explicitly knowing its UserKeyValue.

This property is read-only.

LeafCount

Description

This property returns the number of leaves in the tree. The value range is a number between 0 and 2 billion.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

LeafOnClipboard

Description

Use this property with the HTML Tree Manager to determine if a leaf has been cut and is available to be 'pasted' into tree. This property returns a Boolean value: True if a leaf object exists on the clipboard, False otherwise.

This property is read-only.

LevelCount

Description

This property returns the number of levels defined for the tree. If the tree doesn't have any levels, this property returns zero (0).

See the LevelUse property.

Valid value range is a number between 0 and 99.

This property can only be used with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

See Also

Chapter 33, Tree Classes , LevelUse, page 1731.

Levels

Description

This property returns a reference to a level collection object. This property is used only with trees that have levels: it returns Null for trees that don't have levels.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

See Also

Chapter 33, Tree Classes , Level Collection Methods , page 1678.

LevelUse

Description

This property returns the level use of the tree. The values are:

- "S": Strict levels.
- "L": Loose levels.
- "N": No levels.

If this is a new tree, and you do not set this property, the default value is Strict Levels.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

```
If &MyTree.LevelUse = "S" Then
/* add levels */
End-If;
```

Name

Description

This property returns the name of the tree, as a string. This property is required if you are creating a *new*tree.

Note. Do not use this property to change the name of an existing tree. The change will *not* be reflected in the database. You must use the **Rename** method to change the name of an existing tree.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Example

```
&MyNewTree.Name = "DEPT_SECURITY_ADD";
```

See Also

Chapter 33, Tree Classes , Rename, page 1718.

NodeCount

Description

This property returns the number of nodes for the tree. A valid tree must have at least one node.

Valid value range is a number between 1 and 2 billion.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

NodeOnClipboard

Description

Use this property with the HTML Tree Manager to determine if a node has been cut and is available to be 'pasted' into a tree. This property returns a Boolean value: True if a node object exists on the clipboard, False otherwise.

This property is read-only.

ParentLevel

Description

This property returns the number of the parent branch level in a branched tree. If the tree does *not*have levels or is an unbranched tree, zero (0) is returned. Valid value range is a number between 0 and 100.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

ParentName

Description

This property returns the name of the parent branch in a branched tree, as a string. If the tree does *not*have levels, is an unbranched tree, or if the tree is the root level, Null is returned.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

PerformanceMethod

Description

This property sets the method to use for select and join statements used to generate SQL with the tree. This property is used with nVision layouts and Query.

The values for this property are:

- "L": Suppress Join: Use Literal Values.
- "S": Sub-SELECT Tree Selector.
- "J": Join to Tree Selector.
- "D": User Application Defaults.

If this is a new tree, and you do not set this property, the default value "D" is automatically set.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

PerformanceSelector

Description

This property selects the tree selectors for nVision for the tree. This property is used with nVision layouts and Query.

The values for this property are:

- "D": Dynamic Selectors.
- "S": Static Selectors.

If this is a new tree, and you do not set this property, the default value "S" is automatically set.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

PerformanceSelectorOption

Description

This property selects the tree selector options for nVision for the tree. This property is used with nVision layouts and Query.

The values for this property are:

- "S": Single values.
- "R": Ranges of values (>=…<=).
- "B": Ranges of values (BETWEEN).

If this is a new tree, and you do not set this property, the default value "R" is automatically set.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

SetID

Description

This property returns the SetID for the tree, as a string. This property is used only when the IndirectionMethod property for the tree structure the tree is based on has been set to "S". Otherwise, this property returns Null.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

See Also

Chapter 33, Tree Classes , IndirectionMethod, page 1745.

Status

Description

This property returns the effective status of the tree. The values are:

- "A": Active.
- "I": Inactive .

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

Structure

Description

This property returns a reference to the tree structure of the tree. Before you can use some of the methods or any of the properties of a tree structure, you must **Open** the tree structure.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-only.

StructureName

Description

This property returns the name (*structure ID*) of the tree structure the tree is based on, as a string. This property can be set only with a new tree. You receive an error if you try to change the tree structure for an existing tree.

For a new tree, this property must be set before the UserKeyValue or SetID properties can be set.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

See Also

Chapter 33, Tree Classes , UserKeyValue, page 1735, Chapter 33, Tree Classes , SetID, page 1734.

TotalNodeCount

Description

This property is set by the user to indicate how many nodes a tree may have. Use this property just after creating a new tree to ensure the proper number and spacing of tree nodes. Generally, it isn't necessary to set this value. PeopleSoft Tree Manager automatically calculates the default Node gap size for you.

For example, if you are about to create a large tree through PeopleCode or another programming language, it's a good idea to set this property to the number of nodes you expect the tree to have. It's okay if the tree ends up having fewer nodes. However, you need to specify at least the number of nodes you'll be including: otherwise you won't be able to add additional nodes. The system uses this rough estimate to help allocate space for the tree so that it's stored in the database in an efficient manner.

To get the actual number of nodes a tree contains, use the **NodeCount** property.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

See Also

Chapter 33, Tree Classes , NodeCount, page 1732.

UserKeyValue

Description

Use this property to set the key field value for the Node User Key Field specified from the NodeRecord on the associated tree structure.

This property is used only when the IndirectionMethod property for the tree structure has been set to "U" or "B". Otherwise, this property returns Null.

This property can be used only with an open tree, that is, you must use the **Open** method on the tree before you can use this property.

This property is read-write.

See Also

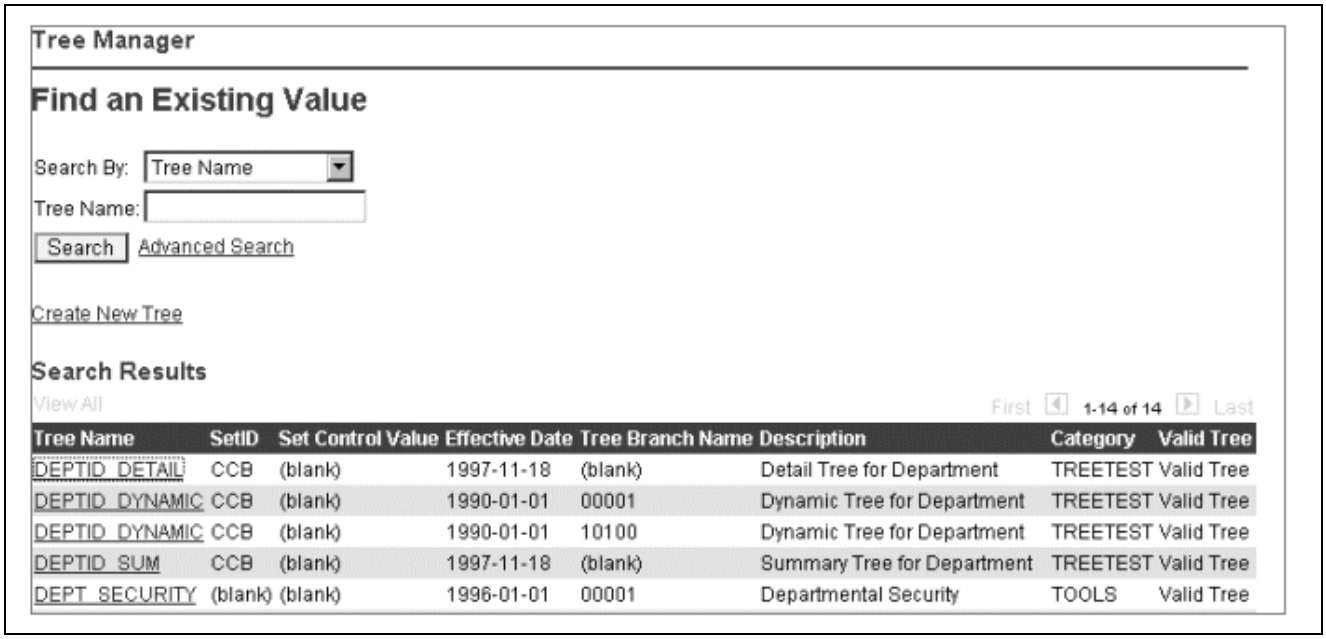
Chapter 33, Tree Classes , NodeRecord, page 1751, Chapter 33, Tree Classes , IndirectionMethod, page 1745.

Tree Collection

A tree collection is returned from the FindTree method used with a session object:

```
&MYCOLL = &SESSION.FindTree("", "", "DEPT", "", "");
```

Most collections have GUI analogies. The following dialog box represents the tree collection.



Tree Collection class GUI representation

All of the tree collection methods return a *closed* tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method. Working with closed trees can improve your performance.

The following example gets all the trees in a tree collection, opens each, and does some processing:

```
Local ApiObject &SESSION;
Local ApiObject &MYTREE;
Local ApiObject &MYTREECOLL;
Local String &SETID, &USERKEY, &TREENAME, &EFFDT, &BRANCHNAME;
Local Boolean &UPDATE;
```

```
&SESSION = %Session;
&MYTREECOLL = &SESSION.FindTree("","","","","");
&MYTREE = &MYTREECOLL.First;
&COUNT = 0;
For &I = 1 to &MYTREECOLL.Count
   &SETID = &MYTREE.KeySetId;
   &USERKEY = &MYTREE.KeyUserKeyValue;
   &TREENAME = &MYTREE.KeyName;
   &EFFDT = &MYTREE.KeyEffDt;
   &BRANCHNAME = &MYTREE.KeyBranchName;
   &UPDATE = True;
   &MYTREE.Open(&SETID, &USERKEYVALUE, &TREENAME, &EFFDT, &BRANCHNAME, &UPDATE);
      /* do processing */
   &MYTREE.Save();
   &MYTREE.Close();
   &COUNT = &COUNT + 1;
   If &COUNT <> &MYTREECOLL.Count Then;
      &MYTREECOLL.Next;
   End-If;
End-For;
```

See Chapter 33, Tree Classes , FindTree, page 1663.

Tree Collection Method

In this section, we discuss the Item method.

Item

Syntax

Item(*SetId*, *UserKeyValue*, *TreeName*, *EffDt*, *BranchName*)

Description

The **Item** method returns the specified tree in the tree collection object executing the method as a *closed* tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

Parameters

Parameter	Description
SetId	Specify the table indirection key for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "S", you must specify a SetID. If the tree structure doesn't have its **IndirectionMethod** specified as "S", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
UserKeyValue	Specify the User Key Value for the tree. This parameter takes a string value. If the tree structure the tree is based on has its **IndirectionMethod** specified as "U" or "B", you must specify a User Key Value. If the tree structure doesn't have its **IndirectionMethod** specified as "U" or "B", you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.
TreeName	Specify the name for this tree. This parameter takes a string value.
EffDt	Specify the effective date for this tree. This parameter takes a string value.
BranchName	Specify the name of the root branch for the tree. This parameter takes a string value. If the tree is unbranched, you must enter a Null string (that is, two quotation marks with no blank space between them ("")) for this parameter.

Note. If the tree is branched and you don't specify the correct branch, a tree is not returned. You can use the **KeyBranchName** property to return the branch name of an unopened tree.

Example

```
&MYTREE = &MYTREECOLL.Item("","","PERSONAL_DATA", "05-05-1997", "");
```

See Also

Chapter 33, Tree Classes , KeyBranchName, page 1729.

Tree Collection Properties

In this section, we discuss the Tree collection properties. The properties are discussed in alphabetical order.

Count

Description

Returns the number of trees for the tree collection object.

First

Description

The **First** property returns the first tree in the tree collection object executing the method as a *closed* tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

Last

Description

The **Last** property returns the last tree in the tree collection object executing the method as a *closed* tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method.

Next

Description

The **Next** property returns the next tree in the tree collection object executing the method as a *closed* tree. A closed tree is a tree object with just the key fields filled in. The rest of the data is not present. To open the tree, you must use the **Open** method. You can use this method only after you have used either the **First** or **Item** properties: otherwise the system doesn't know where to start in the tree.

Tree Structure Class

Tree structure objects are instantiated from the following:

- From a session object with the GetTreeStructure method.
- From a tree object with the Structure property.
- From a Tree Structure Collection and the First, Last, or Next properties or the Item method.

See Chapter 33, Tree Classes , GetTreeStructure, page 1666 and Chapter 33, Tree Classes , Structure, page 1734.

See Chapter 33, Tree Classes , Tree Structure Collection , page 1753.

Tree Structure Class Methods

In this section, we discuss the Tree Structure class methods. The methods are discussed in alphabetical order.

Close

Syntax

```
Close()
```

Description

The **Close** method closes the tree structure object executing the method, freeing the memory associated with that object, and discarding any changes made to the tree structure. The **Close** method can be used only on an open tree structure, not on a closed tree structure. This means you must have opened the tree structure with the **Open** method before you can save it. To save any changes, you must use the **Save** method before using **Close**.

Copy

Syntax

Copy(*FromStructId*, *ToStructId*)

Description

The **Copy** method copies the tree structure specified with *FromStructId to*the tree structure specified with *ToStructId*. Both parameters take string values. The tree structure specified by *ToStructId* must be a *new* tree structure. You receive an error if the tree structure already exists. The tree structure specified by *FromStructId* does *not* have to match the closed tree structure executing the method.

For example, if you created a closed tree structure for the ACCESS_GROUP tree structure, that tree structure wouldn't have to be the *FromStructId*.

```
&MyTreeStruct = &MySession.GetTreeStructure(ACCESS_GROUP);
&MyTreeStruct.Copy(PERSONAL_DATA, PERSONAL_DATA2);
```

The **Copy** method can only be used with a closed tree structure, it cannot be used on an open tree structure. Before you use the **Copy** method, you must explicitly close any open tree structure objects (with the **Close** method.) You receive an error if there are any open tree structures.

To access the new tree structure, you must use the **Open** method.

Create

Syntax

Create(*StructId*, *Type*)

Description

The **Create** method creates a tree structure with the name *StructId*, and of the type *Type*. Both parameters take a string value. The tree structure specified by *StructId* must be a *new* tree structure. You receive an error if the tree structure already exists.

The **Create** method can be used only with a closed tree structure, it cannot be used on an open tree structure. Before you use the **Create** method, you must explicitly close any open tree structure objects (with the **Close** method.) You receive an error if there are any open tree structures.

After you create a new tree structure, you can open it with the **Open** property.

The values for *Type* are:

- "D": create a detail tree structure.
- "S": create a summary tree structure.

The new tree structure must be opened, the required properties set (that is, **Description**, and so on) and then saved (**Save** method) for this structure to be saved to the database.

Delete

Syntax

Delete(*StructId*)

Description

The **Delete** method deletes the tree structure specified by *StructId from the database*. The **Delete** method can be used only with a tree structure identifier, it cannot be used on a fully instantiated, open tree structure. Before you use the **Delete** method, you must explicitly close any open tree structure objects (with the **Close** method.) You receive an error if there are any open tree structures.

The tree structure specified by *StructId* does *not* have to match the tree structure identifier executing the method.

Open

Syntax

Open(*StructId, Update*)

Description

The **Open** method opens the tree structure specified by *StructId*. *StructId* takes a string value. You cannot read or set any properties of a tree structure until after you open it.

When you open a tree structure, you can specify whether you want to open it for update, or in read-only mode, with the *Update* parameter. *Update* takes a Boolean value. Values for *Update* are:

- "True": open tree structure in update mode (read-write).
- "False": open tree structure in read-only mode.

If you try updating or writing to a tree structure opened in read-only mode, you receive an error.

Example

```
&MyTreeStruct = &MySession.GetTreeStructure("ACCESS_GROUP");
&MyTreeStruct.Open("ACCESS_GROUP", True);
```

Rename

Syntax

Rename(*FromStructId, ToStructId*)

Description

The **Rename** method renames the tree structure specified with *FromStructId* to the new name specified by *ToStructId*. Both parameters take a string value. The tree structure specified by *ToStructId* must be a *new* tree structure. You receive an error if the tree structure already exists. The change is automatically reflected in the database.

The **Rename** method can only be used with a closed tree structure, it cannot be used on an open tree structure. Before you use the **Rename** method, you must explicitly close any open tree structure objects (with the **Close** method.) You receive an error if there are any open tree structures.

Save

Syntax

```
Save()
```

Description

The **Save** method writes any changes to the tree structure executing the method to the database.

The **Save** method can be used only on an open tree structure, not on a closed tree structure. This means you must have opened the tree structure with the **Open** method before you can save it.

The tree structure object remains open after executing **Save**. You must execute the **Close** method on the object before it is closed and the memory freed.

Note. If you're calling the Tree API from an Application Engine program, the data won't actually be committed to the database until the Application Engine program performs a COMMIT.

Tree Structure Class Properties

In this section, we discuss the Tree Structure class properties. The properties are discussed in alphabetical order.

Description

Description

This property returns the description for the tree structure, as a string. The description is the text that displays in list boxes or reports.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

```
&MYTREESTRUCT.Description = "Department Security Chart";
```

DetailComponent

Description

This property returns the name of the detail component for the tree structure as a string.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

DetailField

Description

This property returns the name of the detail field for the tree structure, as a string. This property is valid only if you define a detail tree structure. If you create a node-oriented tree structure, this property is not required. This field must exist on the specified DetailRecord.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

The following example sets the **DetailPage, DetailRecord,** and **DetailField** for a tree structure. Note that the names of the page, record, and field are all capitalized: this is required for all PeopleSoft page, record, and field names.

```
&MYTREESTRUCT.DetailPage = "BUS_UNIT_TREE_INV";
&MYTREESTRUCT.DetailRecord = "BUS_UNIT_TBL_IN";
&MYTREESTRUCT.DetailField = "BUSINESS_UNIT";
```

See Also

Chapter 33, Tree Classes , DetailRecord, page 1745.

DetailMenu

Description

This property returns the name of the detail menu for the tree structure, as a string. If the page you use for detail values (DetailPage property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **DetailMenu** and **DetailMenuBar** properties.

This property is valid only if you define a detail tree structure. If you create a node-oriented tree structure, this property isn't required.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

DetailMenuBar

Description

This property returns the name of the detail menu bar for the tree structure, as a string. If the page you use for detail values (DetailPage property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **DetailMenu** and **DetailMenuBar** properties.

This property is valid only if you're define a detail tree structure. If you create a node-oriented tree structure, this property isn't required.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

DetailMenuItem

Description

This property returns the name of the detail menu item for the tree structure as a string.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

DetailMultiNavigate

Description

This property sets or returns a Boolean value indicating whether the detail mulit-navigation tree structure navigation option has been selected or not: True, the Detail Multi-Navigation check box has been selected, False otherwise.

This property is read-write.

See Also

Chapter 33, Tree Classes , NodeMultiNavigate, page 1751, .

DetailPage

Description

This property returns the name of the detail page for the tree structure as a string. If the page you use for detail values (**DetailPage** property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the DetailMenu and DetailMenuBar properties.

This property is valid only if you're defining a detail tree structure. If you're creating a node-oriented tree structure, this property isn't required.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

The following example sets the **DetailPage, DetailRecord,** and **DetailField** for a tree structure. Note that the names of the page, record, and field are all capitalized: this is required for all PeopleSoft page, record, and field names.

```
&MYTREESTRUCT.DetailPage = "BUS_UNIT_TREE_INV";
&MYTREESTRUCT.DetailRecord = "BUS_UNIT_TBL_IN";
```

```
&MYTREESTRUCT.DetailField = "BUSINESS_UNIT";
```

DetailRecord

Description

This property returns the name of the detail record for the tree structure as a string. This property is valid only if youe define a detail tree structure. If you create a node-oriented tree structure, this property isn't required.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

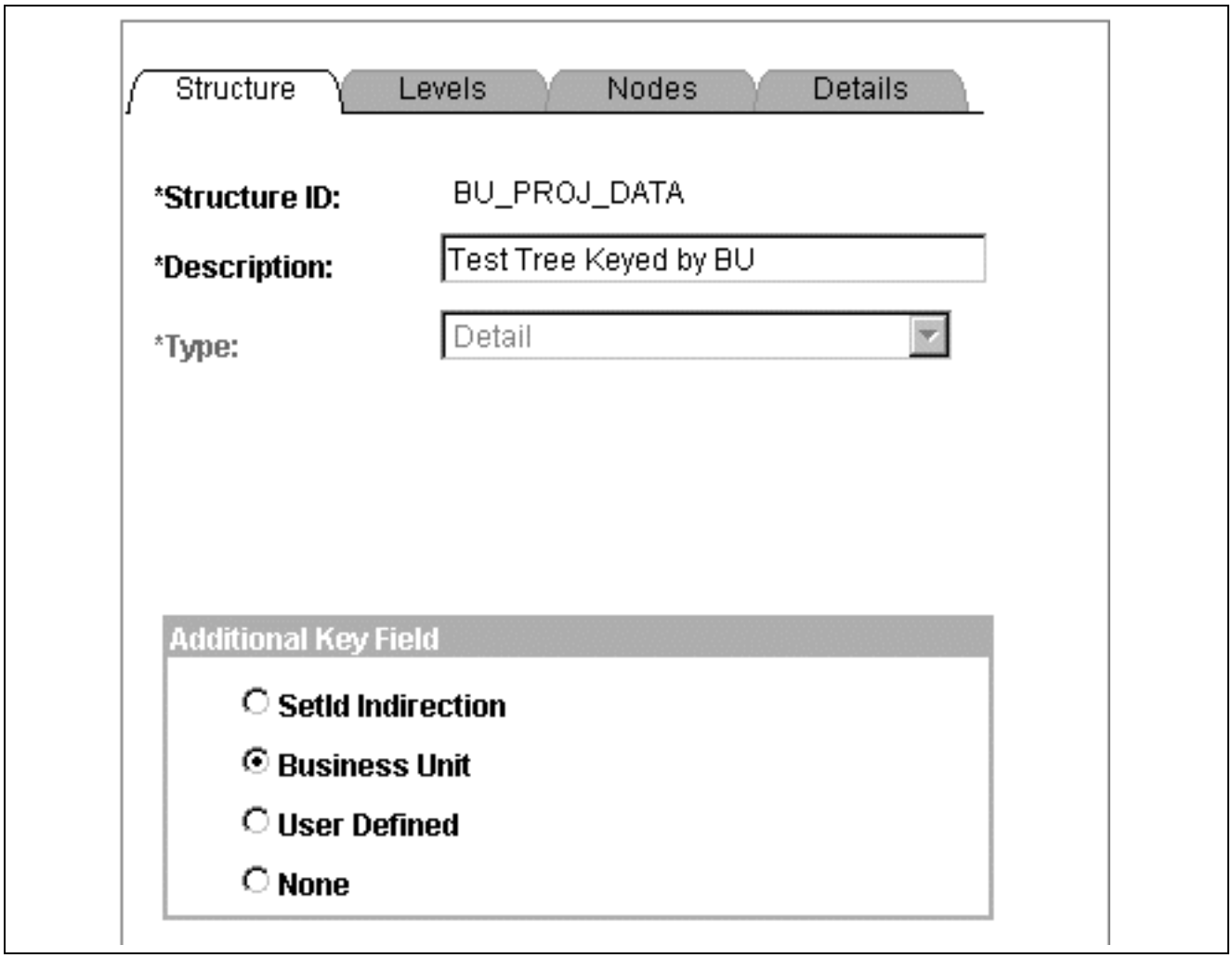
The following example sets the **DetailPage, DetailRecord,** and **DetailField** for a tree structure. Note that the names of the page, record, and field are all capitalized: this is required for all PeopleSoft page, record, and field names.

```
&MYTREESTRUCT.DetailPage = "BUS_UNIT_TREE_INV";
&MYTREESTRUCT.DetailRecord = "BUS_UNIT_TBL_IN";
&MYTREESTRUCT.DetailField = "BUSINESS_UNIT";
```

IndirectionMethod

Description

When you set up a tree structure, you can set the indirection at the bottom of the Structure tab, with the radio buttons in the Additional Key Field.



Structure tab for Tree Structure setup

In the Tree Classes API, you set these same values with the IndirectionMethod property. The values are:

- "S": SetID Indirection.
- "B": Business Unit.
- "U": User Defined Node Key.
- "N": None.

Both the *User Defined Node Key* and the *Business Unit* are the key field set with the NodeRecord property for this tree structure. If *Business Unit* is set, the key field set with the **NodeRecord** property is a business unit.

If you are creating a new tree structure, and you do not explicitly set this property, **None** ("N") is the default value, and is automatically set.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

See Also

Chapter 33, Tree Classes , IndirectionMethod, page 1745, Chapter 33, Tree Classes , NodeRecord, page 1751.

KeyName

Description

This property returns the name of the tree structure as a string.

This property can be used with a closed tree structure. You can use this property to open a tree structure without explicitly knowing its name.

This property is read-only.

Example

```
Local ApiObject &TLIST;
Local ApiObject &TREESTRUCT;
Local ApiObject &MYSESSION;
Local string &NAME;

&MYSESSION = %Session;

If &MYSESSION Then
   /* connection is good */
   &TSLIST = &MYSESSION.FindTreeStructure("");
   For &I = 1 to &TSLIST.Count
      &TREESTRUCT = &TSLIST.Item(&I);
      &NAME = &TREESTRUCT.KeyName;
      &TREESTRUCT.Open(&NAME);
      /* do processing */
   End-For;
Else
   /* do error processing */
End-if;
```

LevelComponent

Description

This property returns the name of the level component for the tree structure as a string.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

LevelMenu

Description

This property returns the name of the level menu for the tree structure as a string. If the page you use for level values (**LevelPage** property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **LevelMenu** and **LevelMenuBar** properties.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

LevelMenuBar

Description

This property returns the name of the level menu bar for the tree structure as a string. If the page you use for level values (**LevelPage** property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **LevelMenu** and **LevelMenuBar** properties.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

LevelMenuItem

Description

This property returns the name of the level menu item for the tree structure as a string.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

LevelPage

Description

This property returns the name of the level page for the tree structure object as a string. If you create a new tree structure, and you do not explicitly set this property, TREE_LEVEL is the default value, and is automatically set.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

LevelRecord

Description

This property returns the name of the level record for the tree structure object as a string. If you create a new tree structure, and you do not explicitly set this property, TREE_LEVEL_TBL is the default value, and is automatically set.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Name

Description

This property returns the name of the tree structure as a string. This is also called the *structure ID*. Use this property to set the name (structure ID) of a tree structure. This property is required if you are creating a *new* tree structure.

Note. Do not use this property to change the name of an existing tree structure. The change will *not* be reflected in the database. You must use the **Rename** method to change the name of an existing tree structure.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

```
&MYTREESTRUCT.Name = "PERSONAL_DATA";
```

See Also

Chapter 33, Tree Classes , Rename, page 1741.

NodeComponent

Description

This property returns the name of the node component for the tree structure as a string.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodeField

Description

This returns the name of the field used for storing information about nodes for the tree structure as a string. If you create a new tree structure, and you do not explicitly set this property, TREE_NODE is the default value, and is automatically set. This field must exist on the specified **NodeRecord**.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Example

```
&MYTREESTRUCT.NodeField = "DEPT_ID";
```

NodeMenu

Description

This property returns the name of the node menu for the tree structure as a string. If the page you use for node values (**NodePage** property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **NodeMenu** and **NodeMenuBar** properties.

If you create a new tree structure, and you do not explicitly set this property, TREE_MANAGER is the default value, and is automatically set.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodeMenuBar

Description

This property returns the name of the node menu bar for the tree structure as a string. If the page you use for node values (**NodePage** property) is part of a component, and you want to have access to the other pages in that group when you edit a node value, enter the name of the menu and menu bar for the component in the **NodeMenu** and **NodeMenuBar** properties.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodeMenuItem

Description

This property returns the name of the node menu item for the tree structure as a string.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodeMultiNavigate

Description

This property sets or returns a Boolean value indicating whether the node mulit-navigation tree structure navigation option has been selected: True, the Node Multi-Navigation check box has been selected, False otherwise.

This property is read-write.

See Also

Chapter 33, Tree Classes , DetailMultiNavigate, page 1744, .

NodePage

Description

This returns the name of the page used for storing information about nodes for the tree structure as a string. If you create a new tree structure, and you do not explicitly set this property, TREE_NODE is the default value, and is automatically set.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

NodeRecord

Description

This returns the name of the record used for storing information about nodes for the tree structure as a string. If you create a new tree structure, and you do not explicitly set this property, TREE_NODE_TBL is the default value, and is automatically set.

If the IndirectionMethod property of this tree structure has been set to "U" or "B", use this property to specify the record that the **NodeUserKeyField** property will use.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

See Also

Chapter 33, Tree Classes , IndirectionMethod, page 1745.

NodeUserKeyField

Description

Use this property to set the key field from the record specified with **NodeRecord** for use with the IndirectionMethod. This property is identical to the UserKeyValue property for a tree.

This property is used only when the **IndirectionMethod** property for the tree structure has been set to "U" or "B". Otherwise, this property returns Null.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

See Also

Chapter 33, Tree Classes , IndirectionMethod, page 1745, Chapter 33, Tree Classes , UserKeyValue, page 1735.

SummarySetId

Description

This property returns the SetID for the detail tree whose detail values the summary tree structure as a string summarizes. This property is valid only for Summary tree structures. Otherwise, this property returns Null.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

SummaryLevelNumber

Description

This property returns the level number for the detail tree whose detail values the summary tree structure summarizes. The level number indicates the level in the detail tree whose nodes the summary tree will use as detail values. The top level in the detail tree is 1, the next is level 2, and so on. Valid values are numbers between 1 and 99.

This property is only valid for Summary tree structures. Otherwise, this property returns Null.

This property can only be used with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

SummaryTreeName

Description

This property returns the name of the detail tree whose detail values the summary tree structure summarizes, as a string. This property is valid only for Summary tree structures. Otherwise, this property returns Null.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

SummaryUserKeyValue

Description

This property returns the name of the tree that the node records the tree structure is summarized to, as a string. This property is valid only for Summary tree structures. This property is used only when the IndirectionMethod property for the tree structure has been set to "U" for *User Defined Keys* or "B" for *Business Unit.* Otherwise, this property returns Null.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

See Also

Chapter 33, Tree Classes , IndirectionMethod, page 1745.

Type

Description

This property returns the type of the tree structure as a string, whether it is a detail or summary tree structure. The values are:

- "D": detail tree structure.
- "S": summary tree structure.

This property can be used only with an open tree structure, that is, you must use the **Open** method on the tree structure before you can use this property.

This property is read-write.

Tree Structure Collection

A tree structure collection is returned from the FindTreeStructure method used with a session object:

```
&MYCOLL = &SESSION.FindTreeStructure("");
```

Most collections have GUI analogies. The following dialog box represents the tree structure collection:

Tree structure collection GUI representation

See Chapter 33, Tree Classes , FindTreeStructure, page 1664.

Tree Structure Collection Method

In this section, we discuss the Item method.

Item

Syntax

Item(*StructureID*)

Description

The **Item** method returns the tree structure specified by *StructureID* in the tree structure collection object executing the method as a closed tree structure. A closed tree structure is a tree structure object with just the key fields filled in. The rest of the data is not present. To open the tree structure, you must use the **Open** method. *StructureID* takes a string value.

Tree Structure Collection Properties

In this section, we discuss the Tree Structure collection properties. The properties are discussed in alphabetical order.

Count

Description

Returns the number of tree structures for the tree structure collection object.

First

Description

The **First** property returns the first tree structure in the tree structure collection object executing the method as a closed tree structure. A closed tree structure is a tree structure object with just the key fields filled in. The rest of the data is not present. To open the tree structure, you must use the **Open** method.

Last

Description

The **Last** property returns the last tree structure in the tree structure collection object executing the method as a closed tree structure. A closed tree structure is a tree structure object with just the key fields filled in. The rest of the data is not present. To open the tree structure, you must use the **Open** method.

Next

Description

The **Next** property returns the next tree structure in the tree structure collection object executing the method as a closed tree structure. A closed tree structure is a tree structure object with just the key fields filled in. The rest of the data is not present. To open the tree structure, you must use the **Open** method. You can use this method only after you have used either the **First** or **Item** properties: otherwise the system doesn't know where to start in the tree.

Traverse Tree Hierarchy Example

The following is an example to transverse a tree hierarchy and write the information to a file.

```
Local ApiObject &TreeObj, &RootNodeObj;
Local any &AnyObj;
Local string &sSetId, &sSetCntrlValue, &sTreeName, &sBranchName;
Local File &MyFile;

Function ProcessNodeChildren(&ParentNodeObj As ApiObject)
```

```
   Local ApiObject &ChildNodeObj, &ChildLeafObj;
   Local boolean &First;

   If &ParentNodeObj.HasChildren Then

      &ChildLeafObj = &ParentNodeObj.FirstChildLeaf;
      While All(&ChildLeafObj)
         &MyFile.WriteLine("Process Leaf Range From : " | ⇒
&ChildLeafObj.RangeFrom);
         &ChildLeafObj = &ChildLeafObj.NextSib;
      End-While;

      If &ParentNodeObj.HasChildNodes Then
         &First = True;
         &ChildNodeObj = &ParentNodeObj.FirstChildNode;
         While &First Or
               &ChildNodeObj.HasNextSib

            If &First Then
               &First = False;
            Else
               &ChildNodeObj = &ChildNodeObj.NextSib;
            End-If;
            &MyFile.WriteLine("Process Leaf Range From : " | &ChildNodeObj.Name);
            ProcessNodeChildren(&ChildNodeObj);

         End-While;
      End-If;
   End-If;
End-Function;

&TreeObj = %Session.GetTree();
&sSetId = "";
&sSetCntrlValue = "";
&sTreeName = "QE_JOBCODES";
&TreeEffDt = Date("1999-01-01");
&sBranchName = "";

&AnyObj = &TreeObj.OPEN(&sSetId, &sSetCntrlValue, &sTreeName, &TreeEffDt, ⇒
&sBranchName, False);
If All(&AnyObj) Then
   &TreeObj = &AnyObj;
End-If;

&MyFile = GetFile("TreeDump.txt", "A");
&RootNodeObj = &TreeObj.FindRoot();
ProcessNodeChildren(&RootNodeObj);
&MyFile.Close();
```

CHAPTER 34

XmlDoc Class

This chapter provides an overview of XmlDoc and XmlNode classes and discusses the following topics:

- When to use an XmlDoc object
- XmlDoc object creation
- XmlNode class considerations
- Error handling
- SoapDoc objects consideration
- Scope of XmlDoc and XmlNode objects
- Data type of an XmlDoc or XmlNode object
- XmlDoc class built-in functions
- XmlDoc methods
- XmlDoc property
- XmlNode methods
- XmlNode properties

Understanding XmlDoc Class

The Extended Markup Language (XML) describes a class of data objects called XML documents. It also partially describes the behavior of computer programs which process them. The XmlDoc class is used to create and manipulate XML data.

The Extended Markup Language (XML) is a method for putting structured data in a text file. Like HTML, XML uses tags, that is, text delimited by brackets (< and >). However, HTML specifies what each tag is, and how it's supposed to be displayed in a browser. XML uses tags only to delimit data. The interpretation of that data is entirely up to the application.

For example, <UL> in HTML specifies an unordered (bulleted) list. However, with XML, it could specify an underlined link.

Each XML document has both a physical and a logical structure:

- Physically, the XML document is composed of units called *entities*. A document begins in a "root" or document entity.

- Logically, the XML document is composed of declarations, elements, comments, character references, and processing instructions

The **CreateXmlDoc** function creates an XmlDoc object. An XmlDoc is composed of XmlNode objects. Each XmlNode represents an XML document *entity*. You can use PeopleCode to create the following types of entities:

- Attribute (specified both by a name and a NamespaceURI)
- CDATA Section
- Comment
- Element
- Entity References
- Process Instruction
- Text

See Also

http://www.w3.org

When to Use an XmlDoc Object

You can use the XmlDoc class to access inbound messages. You can also use the XmlDoc class to create XML messages to be sent out, either synchronously or asynchronously.

All messages must be associated with a message definition. An XmlDoc message must be associated with an *unstructured* message, that is, a message definition that's created without any record definitions.

Use the XmlDoc object if any of the following is true:

- Your message structure doesn't fit the rowset model.
- Your message data doesn't come from database records.
- Your third-party source or target node requires non-XML messages.
- Your message uses the SOAP protocol.

Do *not* use the XmlDoc object if your message contains cookies.

XmlDoc Object Creation

Use the CreateXmlDoc function to create an XmlDoc object. You can create either an empty object, and populate it with data, or you can specify an XML string that is then transformed into an XmlDoc object that you can then manipulate with PeopleCode.

If you're creating an empty XmlDoc object, and you also want to specify a particular document type declaration (DTD) to be used to validate your XML, immediately after you use the CreateXmlDoc function, you should use the **CreateDocumentType** method to specify the DTD, followed by the **CreateDocumentElement** method to associate the DTD with the root entity.

The following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &docTypeNode;
Local XmlNode &rootNode;

&inXMLDoc = CreateXmlDoc("");
&docTypeNode = &inXMLDoc.CreateDocumentType("Personal", "", "Personal.dtd");
&rootNode = &inXMLDoc.CreateDocumentElement("root", "", &docTypeNode);
```

produces the following XML:

```
<?xml version="1.0"?>
<!DOCTYPE Personal SYSTEM "Personal.dtd">
<root/>
```

After you've created your XmlDoc object, use the GenXmlString method to create an XML string. You can then use an Internet Script (iScript) Response object method to send the string as an XML response.

See Also

Chapter 15, Internet Script Classes (iScript) , page 931

XmlNode Class Considerations

In general, the XmlNode Class methods and properties can be broken up into the following categories:

Category	Description
Add	The Addxxx methods add an entity of the specified type to the end of the list of child nodes, and returns a reference to the newly created node.
Insert	The Insertxxx methods insert an entity of the specified type at the specific location and returns a reference to the newly created node.

Category	Description
Get	The Getxxx methods return a reference to the specified entity. The Getxxx methods may return a single reference or an array of references.
Other	The other methods enable you to find a particular entity in an XmlDoc (FindNode), copy data from one node into another, and remove nodes.

Accessing and Traversing an XmlNode Object

If you're creating an XmlDoc object, use the CreateDocumentElement, as well as the different Add and Insert methods to create XmlNode objects. These methods return a reference to the newly created node if successful.

Use the XmlNode properties for traversing the data structure (ParentNode, PreviousSibling, NextSibling, and so on.)

You can also use the ChildNodeCount property for looping through all the child nodes of a node.

Use the NodeType property to get the type of the node (element, processing instruction, comment, and so on).

Error Handling

Use the IsNull property to verify if the XmlNode returned by a method is a valid node. In the following example, a particular node is checked to see if it has another node after it. If it doesn't, a node is added.

```
Local XmlNode &usernode;
Local string &userName;

&usernode = &inXMLDoc.DocumentElement.FindNode("Request/Company/Location/User");

If Not (&MyNode.IsNull) Then
   &userName = &usernode.NodeValue;
Else
   /* Do error processing */
End-If;
```

SOAPDoc Object Considerations

A SOAP Message is an ordinary XML document that consists of the following parts:

- a mandatory SOAP envelope
- an optional SOAP header

- a mandatory SOAP body

Most of the methods and properties that you use with an XmlDoc or XmlNode can also be used with a SOAPDocument. However, there are a few XmlNode methods that cannot be used with a SOAPDocument. The following methods *cannot* be used with a SOAPDocument.

- AddCDataSection
- AddProcessInstruction
- CreateDocumentType
- InsertCDataSection
- InsertProcessInstruction

See Also

Chapter 30, SOAPDoc Class , page 1587

Scope of XmlDoc and XmlNode Objects

XmlDoc and XmlNode objects can be instantiated only from PeopleCode.

This object can be used anywhere you have PeopleCode, that is, in message subscription PeopleCode, Application Engine PeopleCode, record field PeopleCode, and so on.

Data Type of an XmlDoc or XmlNode Object

XmlDoc objects are declared type XmlDoc.

XmlNode objects are declared type XmlNode.

For example:

```
Component XmlDoc &MyDoc;

Local XmlNode &MyNode;
```

Note. XmlNode objects can be declared only as type Local.

XmlDoc Class Built-in Functions

Chapter 1, PeopleCode Built-in Functions, CancelPubHeaderXmlDoc , page 61

Chapter 1, PeopleCode Built-in Functions, CancelPubXmlDoc , page 61

Chapter 1, PeopleCode Built-in Functions, CancelSubXmlDoc , page 62

Chapter 1, PeopleCode Built-in Functions, CreateSOAPDoc , page 109

Chapter 1, PeopleCode Built-in Functions, CreateXmlDoc , page 111

Chapter 1, PeopleCode Built-in Functions, GetMessageXmlDoc , page 255

Chapter 1, PeopleCode Built-in Functions, GetNRXmlDoc , page 261

Chapter 1, PeopleCode Built-in Functions, GetPubHeaderXmlDoc , page 263

Chapter 1, PeopleCode Built-in Functions, GetPubXmlDoc , page 264

Chapter 1, PeopleCode Built-in Functions, GetSubXmlDoc , page 276

Chapter 1, PeopleCode Built-in Functions, GetSyncLogData , page 277

Chapter 1, PeopleCode Built-in Functions, PublishXmlDoc , page 352

Chapter 1, PeopleCode Built-in Functions, ReSubmitPubHeaderXmlDoc , page 374

Chapter 1, PeopleCode Built-in Functions, ReSubmitPubXmlDoc , page 375

Chapter 1, PeopleCode Built-in Functions, ReSubmitSubXmlDoc , page 376

Chapter 1, PeopleCode Built-in Functions, ReValidateNRXmlDoc , page 379

Chapter 1, PeopleCode Built-in Functions, SyncRequestXmlDoc , page 455

Chapter 1, PeopleCode Built-in Functions, Transform , page 476

Chapter 1, PeopleCode Built-in Functions, UpdateXmlDoc , page 492

XmlDoc Methods

In this section, we discuss the XmlDoc methods. The methods are discussed in alphabetical order.

CopyRowset

Syntax

CopyRowset(*&InRowset*[, *MessageName*][, *MessageVersion*])

Description

Use the CopyRowset method to copy rowset data to an XmlDoc object.

Warning! If the XmlDoc is *not* empty, the existing structure and data are replaced with the data and structure from *InRowset* . If *MessageName* is not specified, this function makes the best possible XML structure given the passed rowset. Not passing the message name should only occur when using an unstructured message or when using a standalone rowset. For best performance, PeopleSoft recommends to always specify the message name.

Parameters

Parameter	Description
&InRowset	Specify the variable of an already instantiated and populated rowset to copy data from.
MessageName	Specify the name of the message that the rowset belongs to, as a string. If the message name is not specified, the best possible XML structure is created.
MessageVersion	Specify the message version, as a string. If you do not specify a message version, the default message version is used.

Returns

A Boolean value: True, the rowset data copied successfully, False otherwise.

Example

```
Local XmlDoc &inXMLDoc;
Local boolean &ret;

&inXMLDoc = CreateXmlDoc("");
&ret = &inXMLDoc.CopyRowset(&rs, "<insert message name>", "<insert message⇒
 version");
```

See Also

Chapter 34, XmlDoc Class , CopyToRowset, page 1765.
Chapter 27, Rowset Class , page 1479

CopyToPSFTMessage

Syntax

CopyToPSFTMessage(*targetDoc*, *srcPath*, *targetPath*)

Description

Use the **CopyToPSFTMessage** method to copy an XmlDoc to a PeopleSoft message. The *srcPath* and *targetPath* parameters enable you to map the data of the XmlDoc (using the XmlDoc structure) to the message (following the message structure.)

The XmlDoc object generally has the following structure:

```
<?xml version="1.0"?>
<MESSAGE_NAME>
  <FieldTypes>
    ...
  </FieldTypes>
  <MsgData>
    <Transaction>
      ...
    </Transaction>
  </MsgData>
```

```
</MESSAGE_NAME>
```

Parameters

Parameter	Description
targetDoc	Specify the XmlDoc that contains the message structure.
srcPath	Specify an array of string that contains the mapping path information.
targetPath	Specify an array of string that contains the mapping path information.

Returns

A number. The values are:

Value	Description
0	Ok
1	Missing PSCAMA structure
2	Missing FieldType structure
3	General Error

Example

```
Local XmlDoc &srcDoc, &targetDoc;
Local array of string &srcPath, &targetPath;

&srcPath = CreateArrayRept("", 0);
&targetPath = CreateArrayRept("", 0);


&srcDoc = CreateXmlDoc("");
&targetDoc = CreateXmlDoc("");
&ret = &srcDoc.ParseXmlFromURL("c:/temp/source.xml");
&ret = &targetDoc.ParseXmlFromURL("c:/temp/target.xml");

&srcPath.Push("QE_EMPLOYEE");
&srcPath.Push("QE_EMPLOYEE/QE_ACCOUNT_TBL");
&srcPath.Push("QE_EMPLOYEE/QE_JOBCODE");
&srcPath.Push("QE_EMPLOYEE/DEPTID");
&srcPath.Push("EMAIL_MSG_RCD");
&srcPath.Push("EMAIL_MSG_RCD/EMAIL_FILE_RCD");

&targetPath.Push("QE_EMPLOYEE_TG");
&targetPath.Push("QE_EMPLOYEE_TG/QE_ACCOUNT_TBL_TG");
&targetPath.Push("QE_EMPLOYEE_TG/QE_JOBCODE_TG");
&targetPath.Push("QE_EMPLOYEE_TG/DEPTID_TG");
&targetPath.Push("EMAIL_MSG_RCD_TG");
```

```
&targetPath.Push("EMAIL_MSG_RCD/EMAIL_FILE_RCD_TG");

&ret = &srcDoc.CopyToPSFTMessage(&targetDoc, &srcPath, &targetPath);
```

See Also

Chapter 34, XmlDoc Class , CopyRowset, page 1762, Chapter 34, XmlDoc Class , CopyToRowset, page 1765, Chapter 7, Array Class , page 633.
Chapter 17, Message Class , page 973

CopyToRowset

Syntax

CopyToRowset(*&Rowset, Message_Name,* [*Message_Version*])

Description

Use the CopyToRowset method to copy data from the XmlDoc to an already instantiated rowset.

The rowset must be based on a message object that has the same *structure* as the XmlDoc. That is, if you have a record at level 2 in your message and you want that data, you must have the same level 0 and level 1 records in your message as in your XmlDoc.

For example, suppose your XmlDoc had the following structure:

```
<?xml version="1.0"?>
<QE_SALES_ORDER>
    <QE_ACCT_ID/>
    <QE_ACCOUNT_NAME/>
    <QE_ADDRESS/>
    <QE_PHONE/>
    <QE_FROMROWSET/>
    <QE_TOROWSET/>
    <QE_SEND_SOA_BTN/>
    <QE_SEND_SOS_BTN/>
    <QE_TRAN_SOA_BTN/>
    <QE_SEND_SQ_BTN/>
    <QE_TRAN_SOS_BTN/>
    <QE_TRAN_APCODE_BTN/>
    <QE_TRAN_SPCODE_BTN/>
    <QE_PUBXMLDOC_BTN/>
    <QE_CLEAR_BTN/>
    <DESCRLONG/>
</QE_SALES_ORDER>
```

If you wanted to include the information in the QE_SALES_ORDER record, you would have to have *at least* the following record structure in your message:

```
QE_SALES_ORDER
```

Any records or fields that are in the XmlDoc that aren’t in the message (and vice-versa) are ignored.

Rowsets should be created using the following pseudo code:

```
Local Message &msg;
Local Rowset &rs;
Local XmlDoc &inXMLDoc;
Local boolean &ret;

&msg = CreateMessage(Message.<insert message name>);
&rs = &msg.GetRowset();
&inXMLDoc = CreateXmlDoc("<insert XML structure here>");
&ret = &inXMLDoc.CopyToRowset(&rs, "<insert message name>", "<insert message⇒
 version>");
```

XmlDoc objects have to follow the Peoplesoft message format:

```
<?xml version="1.0"?>
<PSmessage>
    <MsgData>
        <Transaction>
            <Record1 class="R">
                <Field1>xxx</Field1>
                <Field2>yyy</Field2>
                ...
                <Fieldn>nnn</Fieldn>
                <Record2>
                ...
                </Record2>
            </Record1>
        </Transaction>
    </MsgData>
</PSmessage>
```

Warning! If *MessageName* is not specified, this function makes the best possible flat structure. Not passing the message name should only occur when using an unstructured message or when using a standalone rowset. For best performance, PeopleSoft recommends to always specify the message name.

Parameters

Parameter	Description
&InRowset	Specify the variable of an already instantiated rowset to copy data to.
Message_Name	Specify the message name of the message. If the message name is not specified, the best possible flat structure is created.
Message_Version	Specify the message version, as a string. If the message version is not specified, the default message version is used.

Returns

This function always returns a True value, regardless of the success of the operation.

See Also

Chapter 34, XmlDoc Class , CopyRowset, page 1762.
Chapter 27, Rowset Class , page 1479
Chapter 17, Message Class , page 973

CreateDocumentElement

Syntax

CreateDocumentElement(*TagName* [, *NamespaceURI*] [, *&DocType*])

where *NamespaceURI* can have *one* of the following forms:
URL.*URLname*

OR a string URL, such as
http://www.peoplesoft.com/

Description

Use the **CreateDocumentElement** function to create the root element of the document.

For *&DocType* you must specify an already instantiated document type node using CreateDocumentType.

Parameters

Parameter	Description
TagName	Specify the name of the tag to be used for the document element.
NamespaceURI	Specify the URI that contains the Namespaces for the XmlDoc, as a string.
&DocType	Specify the document type node. This must already be instantiated using CreateDocumentType.

Returns

A reference to an XmlNode object representing the root element.

Example

The following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &docTypeNode;
Local XmlNode &rootNode;

&inXMLDoc = CreateXmlDoc("");
&docTypeNode = &inXMLDoc.CreateDocumentType("Personal", "", "Personal.dtd");
&rootNode = &inXMLDoc.CreateDocumentElement("root", "", &docTypeNode);
```

produces the following XML:

```
<?xml version="1.0"?>
<!DOCTYPE Personal SYSTEM "Personal.dtd">
```

```
<root/>
```

See Also

Chapter 34, XmlDoc Class , CreateDocumentType, page 1768.

CreateDocumentType

Syntax

CreateDocumentType(*Name*, *PublicID*, *SystemID*)

Description

Use the **CreateDocumentType** method to create the document type declaration (a document type node) in the XmlDoc.

You can specify only *one* document type node for an XmlDoc.

You need to create a document type node only if you have a document type definition (DTD). You don't need to specify a document type declaration in your XML. Generally, the DTD file is used for validating the XML.

If you do specify this type of node when creating an XmlDoc in PeopleCode, you *must* specify it immediately following the creation statement. Then you must specify the document type when you create the root node. This means your code should start with the **CreateXmlDoc** function, creating the XmlDoc object, then use the **CreateDocumentType** method, followed by the **CreateDocumentElement** method, creating the root node.

Considerations Using Public and System IDs

You can specify both a public and a system ID. However, if you want to use *only* a system ID, you must specify a Null string ("") for the public ID. To use only a public ID, you must specify a Null string ("") for the system ID.

To use the System.dtd as a system ID, you must use the following code:

```
&DocType = &MyDoc.CreateDocumentType("Personal", "", "Personal.dtd");
```

This produces the XML:

```
<!DOCTYPE Personal PUBLIC "Personal.dtd">
```

Parameters

Parameter	Description
Name	Specify the name of the document type declaration as a string.
PublicID	Specify the public ID of the document type declaration as a string.
SystemID	Specify the system ID of the document type declaration as a string.

Returns

A reference to an XmlNode object representing the document type element.

Example

The following example creates a document type declaration using the DTD named "Personal".

```
Local XmlDoc &inXMLDoc;
Local XmlNode &docTypeNode;
Local XmlNode &rootNode;

&inXMLDoc = CreateXmlDoc("");
&docTypeNode = &inXMLDoc.CreateDocumentType("Personal", "", "Personal.dtd");
&rootNode = &inXMLDoc.CreateDocumentElement("root", "", &docTypeNode);
```

This produces the following XML:

```
<?xml version="1.0"?>
<!DOCTYPE Personal SYSTEM "Personal.dtd">
<root/>
```

See Also

Chapter 34, XmlDoc Class , CreateDocumentElement, page 1767, Chapter 1, PeopleCode Built-in Functions, CreateXmlDoc , page 111.

GenFormattedXmlString

Syntax

GenFormattedXmlString()

Description

Use the **GenFormattedXmlString** method to return an XML string representing the XmlDoc object.

The GenFormattedXmlString method produces an XML string with new line characters and indents. If you want an unformatted XML string, that is, with all the data on a single line, use the GenXmlString method instead.

Parameters

None.

Returns

A formatted XML string.

Example

```
Local XmlDoc &inXMLDoc;
Local string &outStr;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><PSMessage/>");
&outStr = &inXMLDoc.GenFormattedXmlString();
```

See Also
Chapter 34, XmlDoc Class , GenXmlString, page 1770.

GenXmlString

Syntax
GenXmlString()

Description

Use the **GenXmlString** method to return an XML string representing the XmlDoc object.

The GenXmlString method produces an XML string with all the data on a single line. If you want a formatted XML string, that is, with new line characters and indents, use the GenFormattedXmlString method instead.

Parameters

None.

Returns

An XML string.

Example

```
Local XmlDoc &inXMLDoc;
Local string &outStr;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><PSMessage/>");
&outStr = &inXMLDoc.GenXmlString();
```

See Also
Chapter 34, XmlDoc Class , GenFormattedXmlString, page 1769.

GetElementsByTagName

Syntax
GetElementsByTagName(*TagName*)

Description

Use the **GetElementsByTagName** method to return an array of XmlNode objects that match the specified tag name.

If you specify an invalid tag name, an array is still returned, however, it has 0 elements in it.

Parameters

Parameter	Description
TagName	Specify the tag name that you want to look for.

Returns

An array of XmlNode objects.

Example

Using the following Xml structure:

```
<?xml version="1.0"?>
<PSmessage>
    <MsgData>
        <Transaction>
            <Record1 class="R">
                <Field1>one</Field1>
                <Field1>two</Field1>
                <Field1>three</Field1>
            </Record1>
        </Transaction>
    </MsgData>
</PSmessage>
```

The following code finds all of the *Field1* elements:

```
Local array of XmlNode &field1List;

&field1List = &inXMLDoc.GetElementsByTagName("Field1");

If &field1List.Len = 0 Then
   /* do error processing */
Else
   /* do processing */
End-If;
```

See Also

Chapter 7, Array Class , page 633.

LoadIBContent

Syntax

LoadIBContent(*Content*[, *RootTagName*])

Description

Use **LoadIBContent** to load data into an XmlDoc object. This function is primarily used for display purposes, the main use is found in the Integration Broker Monitor for viewing message structures.

When *Content* is an XML string, the XmlDoc contains the DOM representation of the XML.

When *Content* is not an XML string, LoadIBContents generates a DOM wrapper.

Parameters

Parameter	Description
Content	Specify the content of the XmlDoc.
RootTagName	Specify the root tag name to be used in the XmlDoc object used as the wrapper, if the data is non-Xml. If a root tag name is not specified, and the data is non-Xml, the contents are not wrapped with the default of root tag name of **PSMessage**.

Returns

This method returns a Boolean value that is always True.

Example

Using the following data:

```
John Q. Public,1234 West Eastland,925-987-0909
Jane Doe,423 Someplace,234-992-9383
```

The following PeopleCode program specifies a root name:

```
Local XmlDoc &inXMLDoc;
Local boolean &ret;
Local string &inStr;

&inStr = "John Q. Public,1234 West Eastland,925-987-0909" | Char(13) | "Jane⇒
 Doe,423 Someplace,234-992-9383";

&inXMLDoc = CreateXmlDoc("");
&ret = &inXMLDoc.LoadIBContent(&inStr, "Root");
```

This produces the following:

```
<?xml version="1.0"?>
<Root>
  <data PsNonXml="Yes">
    <![CDATA[John Q. Public,1234 West Eastland,925-987-0909
Jane Doe,423 Someplace,234-992-9383]]>
  </data>
</Root>
```

The following PeopleCode program does not specify a root name:

```
Local XmlDoc &inXMLDoc;
Local boolean &ret;
Local string &inStr;

&inStr = "John Q. Public,1234 West Eastland,925-987-0909" | Char(13) | "Jane⇒
 Doe,423 Someplace,234-992-9383";
```

```
&inXMLDoc = CreateXmlDoc("");
&ret = &inXMLDoc.LoadIBContent(&inStr);
```

This produces the following:

```
<?xml version="1.0"?>
<PSMessage>
  <data PsNonXml="Yes">
    <![CDATA[John Q. Public,1234 West Eastland,925-987-0909
Jane Doe,423 Someplace,234-992-9383]]>
  </data>
</PSMessage>
```

The following is an example of a valid XML document:

```
Local XmlDoc &inXMLDoc;
Local boolean &ret;
Local string &inStr;

&inStr = "<?xml version='1.0'?><root/>";

&inXMLDoc = CreateXmlDoc("");
&ret = &inXMLDoc.LoadIBContent(&inStr);
```

This produces the following:

```
<?xml version="1.0"?>
<root/>
```

ParseXmlFromURL

Syntax

ParseXmlFromURL(*path*)

Where *path* can have *one* of the following forms—a string URL, containing the filename and extension:
`http://www.peoplesoft.com/filename.ext`

Or an absolute file path, including the filename and extension:
`c:\directory\filename.ext`

Description

Use the **ParseXmlFromURL** method to convert the XML file located at *path* into an XmlDoc object that you can then manipulate using PeopleCode. The XmlDoc object executing the method is populated with the XML string after it's been converted. Any data already existing in the XmlDoc object is overwritten.

Using this method also does basic validation of the XML string, comparing it to the document type declaration (DTD) if the DOCTYPE for the DTD is provided in the XML string.

Parameters

Parameter	Description
path	Specify the URL or file path to the XML that file you want to manipulate.

Returns

A Boolean value: True, the XML file converted successfully and was validated, False otherwise. (If the XML file is not valid, the PeopleCode program terminates.)

Example

The following PeopleCode program loads a file from a file path:

```
Local XmlDoc &inXMLDoc;
Local boolean &ret;

&inXMLDoc = CreateXmlDoc("");
&ret = &inXMLDoc.ParseXmlFromURL("c:\temp\in.xml");
```

The following PeopleCode program loads a file from a URL:

```
Local XmlDoc &inXMLDoc;
Local boolean &ret;

&inXMLDoc = CreateXmlDoc("");
&ret = &inXMLDoc.ParseXmlFromURL("http://www.peoplesoft.com/xmlfile.xml");
```

See Also

Chapter 34, XmlDoc Class , ParseXmlString, page 1774.

ParseXmlString

Syntax

ParseXmlString(*XmlString*)

Description

Use the **ParseXmlString** method to convert the XML string into an XmlDoc object that you can then manipulate using PeopleCode. The XmlDoc object executing the method is populated with the XML string after it's been converted. Any data already existing in the XmlDoc object is overwritten.

Using this method also does basic validation of the XML string, comparing it to the document type declaration (DTD) if the DOCTYPE for the DTD is provided in the XML string.

Parameters

Parameter	Description
XmlString	Specify an XML string that you want to be able to manipulate using PeopleCode.

Returns

A Boolean value: True, the XML string converted successfully and was validated, False otherwise. (If the XML is not valid, the PeopleCode program terminates.)

Example

```
Local XmlDoc &inXMLDoc;
Local boolean &ret;
Local string &str;

&str = "<?xml version='1.0'?><root/>";

&inXMLDoc = CreateXmlDoc("");
&ret = &inXMLDoc.ParseXmlString(&str);

If &ret Then
   /* do processing */
Else
   /* do error processing */
End-If;
```

See Also

Chapter 34, XmlDoc Class , ParseXmlFromURL, page 1773.

XmlDoc Properties

In this section, we discuss the XmlDoc properties. The properties are discussed in alphabetical order.

DocumentElement

Description

This property returns a reference to the root element of the document as an XmlNode.

This property is read-only.

See Also

Chapter 34, XmlDoc Class , XmlNode, page 1776.

IsNull

Description

This property returns a Boolean value, indicating whether the doc is a valid object. This property returns True if the XmlDoc object is valid, False otherwise.

This property is read-only.

XmlNode

Unlike C++ and Java programming model, PeopleCode interface uses the root object XmlNode to represent all 11 different derived objects. An XmlNode object can be defined only as local.

XmlNode Methods

Add methods create a new node of the specific type, append this new node to the child list, and return this newly created node, while Insert methods add the node to the specified position.

In this section, we discuss the XmlNode methods. The methods are discussed in alphabetical order.

AddAttribute

Syntax

```
AddAttribute(Name, Value);
```

Description

Use the **AddAttribute** method to add an attribute to an XmlNode. The new attribute is appended to the list of child nodes. A reference to the newly created attribute is returned.

If you specify an attribute name that already exists, the new value you use *replaces* the existing value, and a reference to the attribute is returned.

Parameters

Parameter	Description
Name	Specify the name of the Attribute that you want to create, as a string.
Value	Specify the value of the Attribute that you want to create, as a string.

Returns

None.

Example

Here is a set of XML response code.

```
<?xml version="1.0"?>
<myroot>
   <postreqresponse>
      <candidate>
         <user>
            <location scenery="great" density="low" blank="eh?"/>

         </user>
      </candidate>
   </postreqresponse>
</myroot>
```

Here's the PeopleCode that builds it.

```
Local XmlDoc &inXMLDoc;
Local XmlNode &postReqNode;
Local XmlNode &candidatesNode;
Local XmlNode &userNode;
Local XmlNode &locationNode;
Local XmlNode &sceneryAtt;
Local XmlNode &densityAtt;
Local XmlNode &blankAtt;
Local boolean &ret;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><myroot/>");

&postReqNode = &inXMLDoc.DocumentElement.AddElement("postreqresponse");
&candidatesNode = &postReqNode.AddElement("candidates");
&userNode = &candidatesNode.AddElement("user");
&locationNode = &userNode.AddElement("location");

&locationNode.AddAttribute("scenery", "great");
&locationNode.AddAttribute("density", "low");
&locationNode.AddAttribute("blank", "eh?");
```

See Also

Chapter 34, XmlDoc Class , IsNull, page 1809.

AddAttributeNS

Syntax

AddAttributeNS(*NamespaceURI*, *AttributeName*, *Value*)

where *NamespaceURI* can have *one* of the following forms:
URL.*URLname*

OR a string URL, such as

```
http://www.peoplesoft.com/
```

Description

Use the **AddAttributeNS** method to add a namespace attribute to an XmlNode. The new attribute is appended to the list of child nodes.

To insert a Namespace attribute at a particular place in the list of nodes, use the InsertAttributeNS method instead.

A reference to the newly created attribute is returned.

If you specify an attribute name that already exists, the new value you use *replaces* the existing value, and a reference to the attribute is returned.

Parameters

Parameter	Description
NamespaceURI	Specify the URI that contains the Namespaces for the XmlDoc, as a string.
AttributeName	Specify the name of the Attribute that you want to create, as a string.
Value	Specify the value of the Attribute that you want to create, as a string.

Returns

None.

Example

```
Local XmlDoc &inXMLDoc;
Local XmlNode &childNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><myroot/>");
&childNode = &inXMLDoc.DocumentElement.AddElement("child");
&childNode.AddAttributeNS("http://www.peoplesoft.com", "scenery", "great");
```

See Also

Chapter 34, XmlDoc Class , IsNull, page 1809.

AddCDataSection

Syntax

AddCDataSection(*Data*)

Description

Use the **AddCDataSection** to add a CDATA section to an XmlNode.

CDATA sections may occur anywhere character data may occur; they are used to escape blocks of text containing characters which would otherwise be recognized as markup.

To insert a CDATA section at a particular place in the list of nodes, use the InsertCDataSection method instead.

A reference to the newly created CDATA section is returned.

Note. You cannot use this method with a SoapDoc object.

Parameters

Parameter	Description
Data	Specify the data to be included in the CDATA section as a string.

Returns

A reference to the newly created CDATA section.

Example

The following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &childNode;
Local XmlNode &cdataNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><myroot/>");
&childNode = &inXMLDoc.DocumentElement.AddElement("child");
&cdataNode = &childNode.AddCDataSection("testing...");
```

produces the following:

```
<?xml version="1.0"?>
<myroot>
  <child>
    <![CDATA[testing...]]>
  </child>
</myroot>
```

See Also

Chapter 34, XmlDoc Class , InsertCDataSection, page 1796.

AddComment

Syntax

AddComment(*Text*)

Description

Use the **AddComment** method to add a comment to an XmlNode. The new attribute is appended to the list of child nodes.

To insert a comment at a particular place in the list of nodes, use the InsertComment method instead.

A reference to the newly created comment is returned.

Parameters

Parameter	Description
Text	Specify the text to be used as the comment, as a string.

Returns

A reference to the newly created comment.

Example

The following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &childNode;
Local XmlNode &commentNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><myroot/>");
&childNode = &inXMLDoc.DocumentElement.AddElement("child");
&commentNode = &childNode.AddComment("This is an example of a comment.");
```

produces the following:

```
<?xml version="1.0"?>
<myroot>
  <child>
    <!--This is an example of a comment.-->
  </child>
</myroot>
```

See Also

Chapter 34, XmlDoc Class , InsertComment, page 1797.

AddElement

Syntax

AddElement(*TagName*)

Description

Use the **AddElement** method to add an element to the XmlNode. The new element is appended to the list of child nodes.

To insert an element at a particular place in the list of nodes, use the InsertElement method instead.

A reference to the newly created element is returned.

Parameters

Parameter	Description
TagName	Specify the tag for the new element that you are adding, as a string.

Returns

A reference to the newly created element.

Example

The following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &childNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><myroot/>");
&childNode = &inXMLDoc.DocumentElement.AddElement("child");
```

produces the following:

```
<?xml version="1.0"?>
<myroot>
  <child/>
</myroot>
```

See Also

Chapter 34, XmlDoc Class , InsertElement, page 1799, Chapter 34, XmlDoc Class , GetElement, page 1793, Chapter 34, XmlDoc Class , GetElementsByTagName, page 1794.

AddElementNS

Syntax

AddElementNS(*NamespaceURI, TagName*)

Where *NamespaceURI* can have *one* of the following forms:
`URL.URLName`

Or a string URL, such as:
`http://www.peoplesoft.com/`

Description

Use **AddElementNS** to add a namespace element to an XmlNode. The new element is appendced to the list of child nodes.

To insert a Namespace element at a particular place in the list of nodes, use the InsertElementNS method instead. A reference to the newly created element is returned. If you specify an element name that already exists, the new value you use replaces the existing value, and a reference to the element is returned.

Parameters

Parameter	Description
NamespaceURI	Specify the URI that contains the Namespaces for the XmlDoc, as a string.
TagName	Specify the name of the Element that you want to create, as a string.

Returns

A reference to the newly created element if successful. If not successful, the IsNull property is set to True.

Example

```
Local XmlDoc &inXMLDoc;
Local XmlNode &childNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><myroot/>");
&childNode = &inXMLDoc.DocumentElement.AddElementNS("http://www.peoplesoft.com",⇒
 "child");
```

AddEntityReference

Syntax

AddEntityReference(*Name*)

Description

Use the **AddEntityReference** method to add an entity reference. An entity reference refers to the content of the named entity. The new entity reference is appended to the list of child nodes. In the generated XML, the entity reference is automatically prefaced with the '&' character and suffixed with a semi-colon.

The named entity must exist in the document type declaration (DTD).

To insert an entity reference at a particular place in the list of nodes, use the InsertEntityReference method instead.

Parameters

Parameter	Description
Name	Specify the name of the entity to which this reference refers to.

Returns

A reference to the newly created entity reference.

Example

The following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &childNode;
```

```
Local XmlNode &entityRef;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><myroot/>");
&childNode = &inXMLDoc.DocumentElement.AddElement("child");
&entityRef = &childNode.AddEntityReference("MyURL");
```

produces the following:

```
<?xml version="1.0"?>
<myroot>
  <child>
&MyURL;  </child>
</myroot>
```

See Also

Chapter 34, XmlDoc Class , InsertEntityReference, page 1801.

AddNode

Syntax

AddNode(*&XmlNode*)

Description

Use the **AddNode** method to add an XmlNode to the XmlDoc. The new node is appended to the list of child nodes.

To insert a node at a particular place in the list of nodes, use the InsertNode method instead.

A reference to the newly created node is returned.

Parameters

Parameter	Description
&XmlNode	Specify an already instantiated XmlNode that you want to add.

Returns

None.

Example

```
Local XmlDoc &inXMLDoc, &firstDoc;
Local XmlNode &childNode;

&firstDoc = CreateXmlDoc("<?xml version='1.0'?><myroot><child><subchild/></child><⇒
/myroot>");
&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&childNode = &firstDoc.DocumentElement.FindNode("child");
&inXMLDoc.DocumentElement.AddNode(&childNode);
```

See Also

Chapter 34, XmlDoc Class , InsertNode, page 1802.

AddProcessInstruction

Syntax

AddProcessInstruction(*Target, Data*)

Description

Use the **AddProcessInstruction** method to add a processing instruction to an XmlNode. The new process instruction is appended to the end of the child list.

To insert a processing instruction at a particular place, use the InsertProcessInstruction method instead.

A reference to the newly created processing instruction is returned.

Note. You cannot use this method with a SoapDoc object.

Parameters

Parameter	Description
Target	Specify the application to which the instruction is directed, as a string.
Data	Specify the data used with the processing instruction.

Returns

A reference to the newly created processing instruction.

Example

The following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &procInst;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><myroot/>");
&procInst = &inXMLDoc.DocumentElement.AddProcessInstruction("xml-stylesheet",⇒
 "href=""book.css"" type=""text/css""");
```

produces the following:

```
<?xml version="1.0"?>
<myroot>
  <?xml-stylesheet href="book.css" type="text/css"?>
</myroot>
```

See Also

Chapter 34, XmlDoc Class , InsertProcessInstruction, page 1804.

AddText

Syntax

AddText(*Data*)

Description

Use the **AddText** method to add a text node to an XmlNode.

Note. A text node is *not* the same as a text declaration. The text declaration is added automatically with the CreateXmlDoc function. A text node just contains text within an XmlDoc.

To insert a text node at a particular place, use the InsertText method instead.

A reference to the newly created text node is returned.

Parameters

Parameter	Description
Data	Specify the data to be used as the text node, as a string.

Returns

A reference to the newly created text node.

Example

The following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &childNode;
Local XmlNode &textNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><myroot/>");
&childNode = &inXMLDoc.DocumentElement.AddElement("child");
&textNode = &childNode.AddText("This is text");
```

produces the following:

```
<?xml version="1.0"?>
<myroot>
  <child>This is text</child>
</myroot>
```

See Also

Chapter 34, XmlDoc Class , InsertText, page 1805.

CopyNode

Syntax

CopyNode(*&Node*)

Description

Use the **CopyNode** method to copy a node specified by *&Node* and *all* of its child nodes into the current node, as a child node. The new node is appended to the end of the list of child nodes. The specified node can belong to a different XmlDoc.

If you just want to copy a top-level node, without copying its children, you can just create a new node with the name of the node you want.

Parameters

Parameter	Description
&Node	Specify an already instantiated node that you want to copy.

Returns

None.

Example

Suppose that your XmlDoc has the following structure:

```
<?xml version="1.0"?>
<myroot>
  <child>
    <subchild/>
  </child>
</myroot>
```

The following PeopleCode copies the node *child* and copies it into another doc.

```
Local XmlDoc &inXMLDoc, &firstDoc;
Local XmlNode &childNode;

&firstDoc = CreateXmlDoc("<?xml version='1.0'?><myroot><child><subchild/></child><⇒
/myroot>");
&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&childNode = &firstDoc.DocumentElement.FindNode("child");
&inXMLDoc.DocumentElement.CopyNode(&childNode);
```

The following is the new structure:

```
<?xml version="1.0"?>
<root>
  <child>
    <subchild/>
```

```
  </child>
</root>
```

See Also

Chapter 34, XmlDoc Class , AddNode, page 1783, Chapter 34, XmlDoc Class , InsertNode, page 1802, Chapter 34, XmlDoc Class , GetChildNode, page 1792, Chapter 34, XmlDoc Class , FindNode, page 1787, Chapter 34, XmlDoc Class , FindNodes, page 1788.

FindNode

Syntax

FindNode(*Path*)

Description

Use the **FindNode** method to return a reference to an XmlNode.

The path is specified as the list of tag names, to the node that you want to find, each separated by a slash (/).

Parameters

Parameter	Description
Path	Specify the tag names up to and including the name of the node that you want returned, each separated by a slash (/). This is known as the XPath query language. See http://www.w3.org/TR/xpath

Returns

An XmlNode matching the path if successful. If not successful, the IsNull property on the XmlNode is set to True.

Example

Suppose your XmlDoc has the following structure:

```
<?xml version="1.0"?>
<myroot>
   <postreqresponse>
      <candidate>
         <user>
            <location scenery="great" density="low" blank="eh?"/>
         </user>
      </candidate>
   </postreqresponse>
</myroot>
```

You want to return a reference to the **location** attribute. Use the following *Path* to do it:

```
&XmlNode = &MyDoc.FindNode("myroot/postreqresponse/candidate/user/location");
```

See Also

Chapter 34, XmlDoc Class , GetChildNode, page 1792, Chapter 34, XmlDoc Class , FindNodes, page 1788.

FindNodes

Syntax

FindNodes(*Path*)

Description

Use the **FindNodes** method to return a reference to an array containing all the XmlNodes that match the path.

The path is specified as the list of tag names, to the node that you want to find, each separated by a slash (/).

Parameters

Parameter	Description
Path	Specify the tag names up to and including the name of the node that you want returned, each separated by a slash (/). This is known as the XPath query language. See http://www.w3.org/TR/xpath

Returns

An array of XmlNode objects. If there is no match for the specified path, an array with 0 elements is returned.

Example

Using the following input:

```
<?xml version="1.0"?>
<PSmessage>
    <MsgData>
        <Transaction>
            <Record1 class="R">
                <Field1>one</Field1>
                <Field1>two</Field1>
                <Field1>three</Field1>
            </Record1>
        </Transaction>
    </MsgData>
</PSmessage>
```

The following PeopleCode program finds all of the *Field1* nodes:

```
Local array of XmlNode &field1List;

&field1List = &inXMLDoc.DocumentElement.FindNodes("MsgData/Transaction/Record1⇒
/Field1");

If &field1List.Len = 0 Then
```

```
   /* do error processing, no nodes returned */
Else
   /* do regular processing */
End-If;
```

See Also

Chapter 34, XmlDoc Class , GetChildNode, page 1792, Chapter 34, XmlDoc Class , FindNode, page 1787, Chapter 7, Array Class , page 633.

GenXmlString

Syntax

GenXmlString()

Description

Use the **GenXmlString** method to generate an XML string of the XmlNode.

Parameters

None.

Returns

An XML String.

Example

```
&MyXML = &MyNode.GenXmlString();
```

See Also

Chapter 34, XmlDoc Class , GenXmlString, page 1789 XmlDoc method

GetAttributeName

Syntax

GetAttributeName(*Index*)

Description

Use the **GetAttributeName** method to return the specified attribute.

Parameters

Parameter	Description
Index	Specify an integer representing the attribute you want to access.

Returns

A string.

Example

```
Local XmlDoc &inXMLDoc;
Local string &attName;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&inXMLDoc.DocumentElement.AddAttribute("name", "Joe");
&inXMLDoc.DocumentElement.AddAttribute("address", "1234 West Eastland");

&attName = &inXMLDoc.DocumentElement.GetAttributeName(2);
```

GetAttributeValue

Syntax

GetAttributeValue({*Name* | *Index*})

Description

Use the **GetAttributeValue** method to return the specific attribute value, given an attribute name. The attribute name is case-sensitive, so you must specify the exact name.

Parameters

Parameter	Description
Name | *Index*	Specify the name or index of the attribute whose value you want to access.

Returns

A string containing the value of the specified attribute.

Example

```
Local XmlDoc &inXMLDoc;
Local string &attName;
Local string &attValue;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&inXMLDoc.DocumentElement.AddAttribute("name", "Joe");
&inXMLDoc.DocumentElement.AddAttribute("address", "1234 West Eastland");

&attName = &inXMLDoc.DocumentElement.GetAttributeName(2);
&attValue = &inXMLDoc.DocumentElement.GetAttributeValue(&attName);
```

GetCDataValue

Syntax

GetCDataValue()

Description

Use the **GetCDataValue** method to return the *value* of the first CDATA section in an XmlNode.

Parameters

None.

Returns

A reference to the value of the first CDATA section as a string.

Example

```
Local XmlDoc &inXMLDoc;
Local XmlNode &dataNode;
Local XmlNode &cdataNode;
Local string &theData;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data");
&cdataNode = &dataNode.AddCDataSection("this is a bunch of text");

&theData = &dataNode.GetCDataValue();
```

See Also

Chapter 34, XmlDoc Class , AddCDataSection, page 1778, Chapter 34, XmlDoc Class , InsertCDataSection, page 1796, Chapter 34, XmlDoc Class , GetCDataValues, page 1791.

GetCDataValues

Syntax

GetCDataValues()

Description

Use the **GetCDataValues** method to return an array of string containing the *values* of the CDATA section in an XmlNode.

Parameters

None.

Returns

An array of string.

Example

```
Local XmlDoc &inXMLDoc;
Local XmlNode &dataNode;
Local XmlNode &cdataNode;
Local array of string &theData;
```

```
&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data");
&cdataNode = &dataNode.AddCDataSection("this is a bunch of text");
&cdataNode = &dataNode.AddCDataSection("more text");
&cdataNode = &dataNode.AddCDataSection("still more text");

&theData = &dataNode.GetCDataValues();
```

See Also

Chapter 34, XmlDoc Class , AddCDataSection, page 1778, Chapter 34, XmlDoc Class , InsertCDataSection, page 1796, Chapter 34, XmlDoc Class , GetCDataValue, page 1790, Chapter 7, Array Class , page 633.

GetChildNode

Syntax

GetChildNode(*index*)

Description

Use the **GetChildNode** method to return the specified child node of an XmlNode.

Parameters

Parameter	Description
Index	Specify an integer representing the child node that you want to access.

Returns

A reference to an XmlNode. If the specified XmlNode isn't found, the IsNull property for the XmlNode is set to True.

Example

```
Local XmlDoc &inXMLDoc;
Local XmlNode &dataNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data2");

&dataNode = &inXMLDoc.DocumentElement.GetChildNode(2);
```

See Also

Chapter 34, XmlDoc Class , FindNode, page 1787, Chapter 34, XmlDoc Class , FindNodes, page 1788.

GetElement

Syntax

GetElement()

Description

Use the **GetElement** method to return the first child element node in the list of child nodes.

Parameters

None.

Returns

A reference to an element.

Example

```
Local XmlDoc &inXMLDoc;
Local XmlNode &dataNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data2");

&dataNode = &inXMLDoc.DocumentElement.GetElement();
```

See Also

Chapter 34, XmlDoc Class , AddElement, page 1780, Chapter 34, XmlDoc Class , InsertElement, page 1799, Chapter 34, XmlDoc Class , GetElements, page 1793, Chapter 34, XmlDoc Class , GetElementsByTagName, page 1794.

GetElements

Syntax

GetElements()

Description

Use the **GetElements** method to return an array of all the XmlNode objects that are elements.

Parameters

None.

Returns

An array of XmlNode of all the element nodes. If there are no element nodes, an array with 0 elements is returned.

Example

```
Local XmlDoc &inXMLDoc;
```

```
Local XmlNode &dataNode;
Local array of XmlNode &dataList;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data2");

&dataList = &inXMLDoc.DocumentElement.GetElements();

If &dataList.Len = 0 Then
   /* do error processing, no nodes returned */
Else
   /* do regular processing */
End-If;
```

See Also

Chapter 34, XmlDoc Class , AddElement, page 1780, Chapter 34, XmlDoc Class , InsertElement, page 1799, Chapter 34, XmlDoc Class , GetElement, page 1793, Chapter 34, XmlDoc Class , GetElementsByTagName, page 1794, Chapter 7, Array Class , page 633.

GetElementsByTagName

Syntax

GetElementsByTagName(*TagName*)

Description

Use the **GetElementsByTagName** method to return an array of XmlNode objects that match the specified tag name.

Parameters

Parameter	Description
TagName	Specify the tag name that you want to look for.

Returns

An array of XmlNode. If you specify an invalid tag name, the returned array has 0 elements.

Example

Using the following input:

```
<?xml version="1.0"?>
<PSmessage>
    <MsgData>
        <Transaction>
            <Record1 class="R">
                <Field1>one</Field1>
                <Field1>two</Field1>
```

```
                <Field1>three</Field1>
            </Record1>
        </Transaction>
    </MsgData>
</PSmessage>
```

The following PeopleCode program finds all *Field1* nodes:

```
Local array of XmlNode &field1List;

&field1List = &inXMLDoc.DocumentElement.GetElementsByTagName("Field1");

If &field1List.Len = 0 Then
   /* do error processing, no nodes returned */
Else
   /* do regular processing */
End-If;
```

See Also

Chapter 34, XmlDoc Class , AddElement, page 1780, Chapter 34, XmlDoc Class , InsertElement, page 1799, Chapter 34, XmlDoc Class , GetElements, page 1793, Chapter 34, XmlDoc Class , GetElement, page 1793, Chapter 7, Array Class , page 633.

GetElementsByTagNameNS

Syntax

GetElementsByTagNameNS(*NamespaceURI, TagName*)

Where *NamespaceURI* can have *one* of the following forms:
URL.*URLName*

Or a string URL, such as:
http://www.peoplesoft.com/

Description

Use the **GetElementsByTagNameNS** method to return an array of XmlNode objects that match the specified namespace and qualified name.

Parameters

Parameter	Description
NamespaceURI	Specify the URI that contains the Namespace for the XmlDoc, as a string.
TagName	Specify the name of the element that you want to retrieve.

Returns

An array of XmlNode. If you specify an invalid NamespaceURI, the returned array has 0 elements.

Example

Using the following input:

```
<?xml version="1.0"?>
<Psmessage xmlns="http://www.peoplesoft.com">
    <MsgData>
        <Transaction>
            <Record1 class="R">
                <Field1>one</Field1>
                <Field1>two</Field1>
                <Field1>three</Field1>
            </Record1>
        </Transaction>
    </MsgData>
</PSmessage>
```

The following PeopleCode program finds all *Field1* nodes:

```
Local array of XmlNode &field1List;

&field1List = &inXMLDoc.DocumentElement.GetElementsByTagNameNS("http:⇒
//www.peoplesoft.com", "Field1");

If &field1List.Len = 0 Then
   /* do error processing, no nodes returned */
Else
   /* do regular processing */
End-If;
```

InsertCDataSection

Syntax

InsertCDataSection(*Data*, *Position*)

Description

Use the **InsertCDataSection** method to insert a CDATA section in an XmlNode, at the location specified by *Position.*

Note. You cannot use this method with a SoapDoc object.

Parameters

Parameter	Description
Data	Specify the data for the CDATA section as a string.
Position	Specify where you want to insert the CDATA Section, as a number.

Returns

A reference to the newly created CDATA section.

Example

Using the following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &dataNode;
Local XmlNode &cdataNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data");
&cdataNode = &dataNode.AddCDataSection("this is a bunch of text");
&cdataNode = &dataNode.AddCDataSection("still more text");

&cdataNode = &dataNode.InsertCDataSection("more text", 2);
```

This is the XML document before the insert:

```
<?xml version="1.0"?>
<root>
  <data>
    <![CDATA[this is a bunch of text]]>
    <![CDATA[still more text]]>
  </data>
</root>
```

This is the XML document after the insert:

```
<?xml version="1.0"?>
<root>
  <data>
    <![CDATA[this is a bunch of text]]>
    <![CDATA[more text]]>
    <![CDATA[still more text]]>
  </data>
</root>
```

See Also

Chapter 34, XmlDoc Class , AddCDataSection, page 1778, Chapter 34, XmlDoc Class , GetCDataValue, page 1790, Chapter 34, XmlDoc Class , GetCDataValues, page 1791.

InsertComment

Syntax

InsertComment(*Data, Position*)

Description

Use the **InsertComment** method to insert a comment in an XmlNode at the location specified by *Position*.

Parameters

Parameter	Description
Data	Specify the data for the comment as a string.
Position	Specify where you want to insert the comment, as a number.

Returns

A reference to the newly created comment.

Example

Using the following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &dataNode;
Local XmlNode &commentNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data");
&commentNode = &dataNode.AddComment("this is a comment");
&commentNode = &dataNode.AddComment("still another comment");

&commentNode = &dataNode.InsertComment("more comments", 2);
```

This is the XML document before the insert:

```
<?xml version="1.0"?>
<root>
  <data>
    <!--this is a comment-->
    <!--still another comment-->
  </data>
</root>
```

This is the XML document after the insert:

```
<?xml version="1.0"?>
<root>
  <data>
    <!--this is a comment-->
    <!--more comments-->
    <!--still another comment-->
  </data>
</root>
```

See Also

Chapter 34, XmlDoc Class , AddComment, page 1779.

InsertElement

Syntax

InsertElement(*TagName*, *Position*)

Description

Use the **InsertElement** method to insert an element in an XmlNode at the location specified by *Position*.

Parameters

Parameter	Description
TagName	Specify the tagname for the element as a string.
Position	Specify where you want to insert the element, as a number.

Returns

A reference to the newly created element.

Example

Using the following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &dataNode;
Local XmlNode &elementNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data");
&elementNode = &dataNode.AddElement("first");
&elementNode = &dataNode.AddElement("second");

&commentNode = &dataNode.InsertElement("third", 2);
```

This is the XML document before the insert:

```
<?xml version="1.0"?>
<root>
  <data>
    <first/>
    <second/>
  </data>
</root>
```

This is the XML document after the insert:

```
<?xml version="1.0"?>
<root>
  <data>
    <first/>
    <third/>
    <second/>
  </data>
</root>
```

See Also

Chapter 34, XmlDoc Class , AddElement, page 1780.

InsertElementNS

Syntax

InsertElementNS(*NamespaceURI, TagName, Position*)

where *NamespaceURI* can have *one* of the following forms:
`URL.URLname`

OR a string URL, such as
`http://www.peoplesoft.com/`

Description

Use **InsertElementNS** to insert a new element in an XmlNode, at the location specified by *Position*.

Parameters

Parameter	Description
NamespaceURI	Specify the URI that contains the Namespaces for the XmlDoc.
TagName	Specify the name of the element that you want to create, as a string.
Position	Specify where you want to insert the element, as a number.

Returns

A reference to the newly created element.

Example

Using the following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &dataNode;
Local XmlNode &elementNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root xmlns='http:⇒
//www.peoplesoft.com'/>");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data");
```

```
&elementNode = &dataNode.AddElement("first");
&elementNode = &dataNode.AddElement("second");

&elementNode = &dataNode.InsertElementNS("http://www.peoplesoft.com", "third", 2);
```

This is the XML document before the insert:

```
<?xml version="1.0"?>
<root xmlns="http://www.peoplesoft.com">
  <data>
    <first/>
    <second/>
  </data>
</root>
```

This is the XML document after the insert:

```
<?xml version="1.0"?>
<root xmlns="http://www.peoplesoft.com">
  <data>
    <first/>
    <third/>
    <second/>
  </data>
</root>
```

InsertEntityReference

Syntax

InsertEntityReference(*Name*, *Position*)

Description

Use the **InsertEntityReference** method to add an entity reference. An entity reference refers to the content of the named entity.

Parameters

Parameter	Description
Name	Specify the name of the entity to which this reference refers to.
Position	Specify where you want to insert the entity reference, as a number.

Returns

A reference to the newly created entity reference.

Example

Using the following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &dataNode;
Local XmlNode &entityRef;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&dataNode = &inXMLDoc.DocumentElement.AddElement("data");
&entityRef = &dataNode.AddEntityReference("first");
&entityRef = &dataNode.AddEntityReference("second");

&entityRef = &dataNode.InsertEntityReference("third", 2);
```

This is the XML document before the insert:

```
<?xml version="1.0"?>
<root>
  <data>
&first;&second;  </data>
</root>
```

This is the XML document after the insert:

```
<?xml version="1.0"?>
<root>
  <data>
&first;&third;&second;  </data>
</root>
```

See Also

Chapter 34, XmlDoc Class , AddEntityReference, page 1782.

InsertNode

Syntax

InsertNode(*&NewNode*, {*&RefNode* | *Position*})

Description

Use the **InsertNode** method to insert a new node. The node is inserted either before the node specified by *&RefNode*, or at the location specified by *Position*.

&NewNode refers to an already instantiated XmlNode object. You must create the node before you can insert it.

&RefNode refers to an already instantiated XmlNode object.

Parameters

Parameter	Description
&NewNode	Specify the name of an already instantiated XmlNode that you want to insert.
&RefNode | *Position*	Specify either the name of an existing node or the location where you want the node to be inserted.

Returns

A reference to the newly inserted node.

Example

The following PeopleCode program inserts a node using a position:

```
Local XmlDoc &inXMLDoc, &firstDoc;
Local XmlNode &childNode;

&firstDoc = CreateXmlDoc("<?xml version='1.0'?><myroot><third><child/></third><⇒
/myroot>");
&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root><first/><second/></root>");
&childNode = &firstDoc.DocumentElement.FindNode("third");

&inXMLDoc.DocumentElement.InsertNode(&childNode, 2);
```

The following PeopleCode program inserts a node using a reference node:

```
Local XmlDoc &inXMLDoc, &firstDoc;
Local XmlNode &childNode, &secondNode;

&firstDoc = CreateXmlDoc("<?xml version='1.0'?><myroot><third><child/></third><⇒
/myroot>");
&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root><first/><second/></root>");
&childNode = &firstDoc.DocumentElement.FindNode("third");
&secondNode = &inXMLDoc.DocumentElement.FindNode("second");

&inXMLDoc.DocumentElement.InsertNode(&childNode, &secondNode);
```

The following is the XML document before the insert:

```
<?xml version="1.0"?>
<root>
  <first/>
  <second/>
</root>
```

The following is the XML document after the insert:

```
<?xml version="1.0"?>
```

```
<root>
  <first/>
  <third>
    <child/>
  </third>
  <second/>
</root>
```

See Also

Chapter 34, XmlDoc Class , AddNode, page 1783.

InsertProcessInstruction

Syntax

InsertProcessInstruction(*Target*, *Data*, *Position*)

Description

Use the **InsertProcessInstruction** method to insert a processing instruction to an XmlNode at the location specified by *Position.*

Note. Youcannot use this method with a SoapDoc object.

Parameters

Parameter	Description
Target	Specify the application to which the instruction is directed, as a string.
Data	Specify the data to be used with the instruction.
Position	Specify where you want to insert the process instruction, as a number.

Returns

A reference to the newly created processing instruction.

Example

Using the following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &procInst;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&procInst = &inXMLDoc.DocumentElement.AddProcessInstruction("first", "firstvalue");
&procInst = &inXMLDoc.DocumentElement.AddProcessInstruction("second",⇒
 "secondvalue");

&procInst = &inXMLDoc.DocumentElement.InsertProcessInstruction("third",⇒
 "thirdvalue", 2);
```

This is the XML document before the insert:

```
<?xml version="1.0"?>
<root>
  <?first firstvalue?>
  <?second secondvalue?>
</root>
```

This is the XML document after the inser:

```
<?xml version="1.0"?>
<root>
  <?first firstvalue?>
  <?third thirdvalue?>
  <?second secondvalue?>
</root>
```

See Also

Chapter 34, XmlDoc Class , AddProcessInstruction, page 1784.

InsertText

Syntax

InsertText(*Data, Position*)

Description

Use the **InsertText** method to insert a text node. The node is inserted at the location indicated by *Position*.

Note. A text node is *not* the same as a text declaration. The text declaration is added automatically with the CreateXmlDoc function. A text node just contains text within an XmlDoc.

Parameters

Parameter	Description
Data	Specify the data to be used for the text node, as a string.
Position	Specify where you want to insert the text ndoe, as a number.

Returns

A reference to the newly created text node.

Example

Using the following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
```

```
Local XmlNode &textNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&textNode = &inXMLDoc.DocumentElement.AddText("first text");
&textNode = &inXMLDoc.DocumentElement.AddText("second text");

&textNode = &inXMLDoc.DocumentElement.InsertText("third text", 2);
```

This is the XML document before the insert:

```
<?xml version="1.0"?>
<root>
first textsecond text</root>
```

This is the XML document after the insert:

```
<?xml version="1.0"?>
<root>
first textthird textsecond text</root>
```

See Also

Chapter 34, XmlDoc Class , AddText, page 1785.

RemoveAllChildNode

Syntax

RemoveAllChildNode()

Description

Use the **RemoveAllChildNode** method to remove all child nodes for an XmlNode.

Parameters

None.

Returns

None.

Example

Using the following PeopleCode program:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &elementNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&elementNode = &inXMLDoc.DocumentElement.AddElement("first");
&elementNode = &inXMLDoc.DocumentElement.AddElement("second");
&elementNode = &inXMLDoc.DocumentElement.AddElement("third");
```

```
&inXMLDoc.DocumentElement.RemoveAllChildNode();
```

This is the XML document before the removal:

```
<?xml version="1.0"?>
<root>
  <first/>
  <second/>
  <third/>
</root>
```

This is the XML document after the removal:

```
<?xml version="1.0"?>
<root/>
```

See Also

Chapter 34, XmlDoc Class , AddNode, page 1783, Chapter 34, XmlDoc Class , InsertNode, page 1802, Chapter 34, XmlDoc Class , GetChildNode, page 1792, Chapter 34, XmlDoc Class , FindNode, page 1787, Chapter 34, XmlDoc Class , FindNodes, page 1788, Chapter 34, XmlDoc Class , RemoveChildNode, page 1807.

RemoveChildNode

Syntax

RemoveChildNode({*Position* | *Node*})

Description

Use the **RemoveChildNode** method to remove the child node in an XmlNode specified by *Node*, or located at *Position*.

Parameters

Parameter	Description
Position | *Node*	Specify the child node that you want to delete, either using its position number or its name.

Returns

A reference to the removed XmlNode.

Example

The following PeopleCode program uses a position:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &elementNode, &removedNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
```

```
&elementNode = &inXMLDoc.DocumentElement.AddElement("first");
&elementNode = &inXMLDoc.DocumentElement.AddElement("second");
&elementNode = &inXMLDoc.DocumentElement.AddElement("third");

&removedNode = &inXMLDoc.DocumentElement.RemoveChildNode(2);
```

The following PeopleCode program uses a reference node:

```
Local XmlDoc &inXMLDoc;
Local XmlNode &elementNode, &removedNode;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&elementNode = &inXMLDoc.DocumentElement.AddElement("first");
&elementNode = &inXMLDoc.DocumentElement.AddElement("second");
&elementNode = &inXMLDoc.DocumentElement.AddElement("third");

&elementNode = &inXMLDoc.DocumentElement.FindNode("second");
&removedNode = &inXMLDoc.DocumentElement.RemoveChildNode(&elementNode);
```

This is the XML document before the removal:

```
<?xml version="1.0"?>
<root>
  <first/>
  <second/>
  <third/>
</root>
```

This is XML document after the removal:

```
<?xml version="1.0"?>
<root>
  <first/>
  <third/>
</root>
```

See Also

Chapter 34, XmlDoc Class , AddNode, page 1783, Chapter 34, XmlDoc Class , InsertNode, page 1802, Chapter 34, XmlDoc Class , GetChildNode, page 1792, Chapter 34, XmlDoc Class , FindNode, page 1787, Chapter 34, XmlDoc Class , FindNodes, page 1788, Chapter 34, XmlDoc Class , RemoveAllChildNode, page 1806.

XmlNode Properties

The following are the XmlNode properties.

AttributesCount

Description

This property returns the number of attributes for an XmlNode, as a number.

This property is read-only.

Example

```
Local XmlDoc &inXMLDoc;

&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&inXMLDoc.DocumentElement.AddAttribute("first", "John");
&inXMLDoc.DocumentElement.AddAttribute("last", "Public");
&inXMLDoc.DocumentElement.AddAttribute("address", "1234 West Eastland");

For &i = 1 To &inXMLDoc.DocumentElement.AttributesCount
   /* do some processing of the attributes here */
End-For;
```

ChildNodeCount

Description

This property returns the number of child nodes for an XmlNode, as a number.

This property is read-only.

Index

Description

This property returns the location (or position) of a node in the parent's child list, as a number. The root node always returns a 1.

This property is read-only.

IsNull

Description

This property returns a Boolean value, indicating whether the node is a valid node. Use this property to verify the value of the add or insert XmlNode methods.

This property is read-only.

Example

```
Local XmlDoc &inXMLDoc;
Local XmlNode &elementNode;
```

```
&inXMLDoc = CreateXmlDoc("<?xml version='1.0'?><root/>");
&elementNode = &inXMLDoc.DocumentElement.AddElement("first");

&elementNode = &inXMLDoc.DocumentElement.FindNode("second");

If &elementNode.IsNull Then
   /* do some error processing here, will not find "second" */
End-If;
```

LocalName

Description

This property returns the local part of a namespace as a string. The local part is the second part of the namespace entry. If you want access to the first part of a namespace, use the Prefix property.

This property is read-only.

See Also

Chapter 34, XmlDoc Class , Prefix, page 1812.

NamespaceURI

Description

This property returns a reference to the namespace URI as a string.

This property is read-only.

NextSibling

Description

This property returns the next child node in an XmlNode, as an XmlNode.

Use the IsNull property to verify that the XmlNode that is returned is valid.

This property is read-only.

See Also

Chapter 34, XmlDoc Class , IsNull, page 1809.

NodeName

Description

This property returns the name of the XmlNode as a string. You can use this property to rename an XmlNode.

This property is read-write.

NodePath

Description

This property returns the path of the XmlNode, from the root.

This property is read-only.

NodeType

Description

This property returns the type of the XmlNode. You can use either a numeric value or a constant. The values are:

Numeric Value	Constant Value	Description
1	%ElementNode	Element node
2	%AttributeNode	Attribute node
3	%TextNode	Text node
4	%CDataSectionNode	CData section node
5	%EntityReferenceNode	Entity reference node
6	%EntityNode	Entity node
7	%ProcessingInstructionNode	Processing instruction node
8	%CommentNode	Comment node
9	%DocumentNode	Document node
10	%DocumentTypeNode	Document type node
11	%NotationNode	Notation node
12	%XmlDeclNode	XML DECL node

NodeValue

Description

This property returns the value of the XmlNode, as a string.

This property is read-write.

ParentNode

Description

This property returns a reference to the parent node for the XmlNode, as an XmlNode object.

Use the IsNull property to verify that the XmlNode that is returned is valid.

This property is read-only.

See Also

Chapter 34, XmlDoc Class , IsNull, page 1809.

Prefix

Description

This property returns the prefix part of a namespace as a string. The prefix is the first part of the namespace entry. If you want access to the second part of a namespace, use the LocalName property.

This property is read-only.

See Also

Chapter 34, XmlDoc Class , LocalName, page 1810.

PreviousSibling

Description

This property returns a reference to the previous node in the XmlNode, as an XmlNode object.

Use the IsNull property to verify that the XmlNode that is returned is valid.

This property is read-only.

See Also

Chapter 34, XmlDoc Class , IsNull, page 1809.

APPENDIX A

Appendix A: Cheat Sheets

This chapter discusses the following topics:

- Class list cheat sheet
- Session class methods and properties
- Internet script classes methods and properties
- Mapping of functions to methods
- Mapping of old names to new names
- PeopleCode syntax cheat sheet

Class List Cheat Sheet

The following tables contain all of the classes, their functions, properties, and methods introduced for PeopleTools.

A method marked with an asterisk (*) is the default method for that class.

Properties that are read-only are marked with RO (read-only). If a property isn't marked RO, it's read-write.

Throughout this section, we use typographical conventions to distinguish between different elements of the PeopleCode language, such as bold to indicate function names, italics for arguments, and so on.

See Also

Chapter 1, PeopleCode Built-in Functions, PeopleCode Typographical Conventions, page 1

Summary of PeopleCode Classes

This table lists the PeopleCode classes, their functions, methods and properties, and what each object returns.

Class Name	Functions, Methods and Properties	Returns
AESection	*Functions*	
	GetAESection(*applid, ae_section* [, *effdt*])	AESection object

Class Name	Functions, Methods and Properties	Returns
AESection	*Methods*	
	AddStep(*ae_step_name* [,*NewStepName*])	
	Close()	
	Open(*ae_applid*, *ae_section*, [*effdt*])	AESection object
	Save()	
	SetSQL(*action_type_string*, *string*)	
	SetTemplate(*ae_applid*, *ae_section*)	
AESection	*Property*	
	IsOpen	Boolean, RO
Application	*Functions*	
	Class *classname* **[Extends** *Classname*] [Method_declarations] [Property_declarations] [**Private** [Method_declarations] [Instance_declarations] [Constant_declaration]] **End-Class**	
	Get *PropertyName* *StatementList* **End-Get**	Depends on assignment of *StatementList*
	Import *PackageName*:[*PackageName*. . .:]{*Classname* | *}	

Class Name	Functions, Methods and Properties	Returns
	Method *MethodName* *StatementList* **End-Method**	Depends on method
	Set *PropertyName* *StatementList* **End-Set**	
Array	*Functions*	
	CreateArray(*paramlist*)	Array object
	CreateArrayAny([*paramlist*])	Array object
	CreateArrayRept(*val*, *count*)	Array object
	Split(*string*, *separator*)	Array object
Array	*Methods*	
	Clone()	Array object
	Find(*value*)	Index of an element
	Get(*index*)	Element of array
	Join([*separator* [, *arraystart*, *arrayend*]])	String
	Next(*&index*)	Boolean
	Pop()	Value of last array element
	Push(*paramlist*)	
	Replace(*start*, *length*, [*paramlist*])	

Class Name	Functions, Methods and Properties	Returns
	Reverse()	
	Set(*index*)	
	Shift()	Value of first array element
	Sort([*order*])	
	Subarray(*start*, *length*)	Array object
	Substitute(*old_val*, *new_val*)	
	Unshift(*paramlist*)	
Array	*Properties*	
	Dimension	Number, RO
	Len	Number
Business Interlink	*Function*	
	GetBIDoc([*XMLString*])	BIDoc object
	GetInterlink(INTERLINK.*name*)	Business Interlink object
Business Interlink	*Methods*	
	AddInputRow(*inputname*, *value*)	Boolean
	BulkExecute(RECORD.*inputrec* [, RECORD.*outputrec*] [, *user_process_inst* | *user_operid*])	Number
	Clear()	
	Execute()	Number

Class Name	Functions, Methods and Properties	Returns
	FetchIntoRecord (RECORD.*recname*, [, *user_process_inst* | *user_operid*])	Number
	FetchIntoRowset(&*Rowset*)	Number
	FetchNextRow(*outputname*, *value*)	Output row object
	GetFieldCount()	Number
	GetFieldType(*index*)	Number
	GetFieldValue(*index*)	String
	InputRowset(&*Rowset*)	Number
	MoveFirst()	Boolean
	MoveNext()	Boolean
Business Interlink	*Property*	
	StopAtError	Boolean
Business Interlink	*Configuration Parameters*	
	URL	String
	Configuration parameters can be accessed like RW properties. Values depend on type of Interlink plug-in.	
Business Interlink BIDoc	*Methods*	
	AddAttribute(*attributename*, *attributevalue*)	Number
	AddComment(*comment*)	Number

Class Name	Functions, Methods and Properties	Returns
	AddProcessingInstruction (*instruction*)	Number
	AddText(*text*)	Number
	CreateElement(*elementname*)	BIDoc object
	GenXMLString()	String
	GetAttributeName(*attributenumber*)	String
	GetAttributeValue({*attributenumber* | *attributename*})	String
	GetNode({*nodenumber* | *nodename*})	BIDoc object
Business Interlink BIDoc	*Properties*	
	AttributeCount	Number, RO
	ChildNodeCount	Number, RO
	NodeName	String, RO
	NodeType	Number, RO
	NodeValue	String, RO
Chart	*Functions*	
	CreateObject("Chart")	Chart object
	GetChart(*RecordName.FieldName*)	Chart object
Chart data	*Methods*	
	Refresh()	

Class Name	Functions, Methods and Properties	Returns
	SetData({*Record_Name* | *&Rowset*})	
	SetDataSeries(*Record_Name.Field_Name*)	
	SetDataHints(*Record_Name.Field_Name*)	
	SetDataURLs(*URLString*)	
	SetDataXAxis(*Record_Name.Field_Name*)	
	SetDataYAxis(*Record_Name.Field_Name*)	
Chart data	*Properties*	
	DataStartRow	Number
	DataWidth	Number
	ImageMap	image map, RO
Chart Appearance	*Properties*	
	GridLines	String
	Height	Number
	IsDrillable	Boolean
	IsPlainImage	Boolean
	LineType	String
	RotationAngle	Number

Class Name	Functions, Methods and Properties	Returns
	Style	String
	StyleSheet	String
	Type	String
	Width	Number
Chart Color	*Methods*	
	SetColorArray(*&Array_of_Color*)	
	SetDataColor(*Record_Name.Field_Name*)	
	SetOLDataColor(*Record_Name.Field_Name*)	
Chart Axis	*Methods*	
	SetXAxisLabels(*&Array_of_Any*)	
	SetYAxisLabels(*&Array_of_Any*)	
Chart Axis	*Properties*	
	XAxisCross	String
	XAxisLabelOrient	String
	XAxisStyle	String
	YAxisLabelOrient	String
	YAxisMax	Number
	YAxisMin	Number

Class Name	Functions, Methods and Properties	Returns
	YAxisStyle	String
	YAxisTicks	Number
Chart Title	*Properties*	
	MainTitle	String
	MainTitleOrient	String
	MainTitleStyle	String
	XAxisTitle	String
	XAxisTitleOrient	String
	XAxisTitleStyle	String
	YAxisTitle	String
	YAxisTitleOrient	String
	YAxisTitleStyle	String
Chart Legend	*Method*	
	SetLegend(*&Array_of_String*)	
Chart Legend	*Properties*	
	HasLegend	Boolean
	LegendMaxEntries	Number
	LegendPosition	String
	LegendStyle	String

Class Name	Functions, Methods and Properties	Returns
Chart Overlay	*Methods*	
	SetOLData({*Record_Name* | *&Rowset*})	
	SetOLDataCol(*Record_Name.Field_Name*)	
	SetOLDataSeries(*Record_Name.Field_Name*)	
	SetOLDataURLs(*URLString*)	
	SetOLDataXAxis(*Record_Name.Field_Name*)	
Chart Overlay	*Properties*	
	OLLineType	
	OLType	
Component Interface	See Appendix A: Cheat Sheets , Component Interface Class Methods and Properties , page 1862.	
Exception	*Functions*	
	CreateException(*message_set*, *message_num*, *default_txt [*, *subslist*])	Exception object
	Throw *Expression*	
	try *Protected StatementList* catch *QualifiedID &Id* *StatementList* end-try	

Class Name	Functions, Methods and Properties	Returns
Exception	*Methods*	
	GetSubstitution(*Index*)	String
	Output()	Depends on context
	SetSubstitution(*Index*, *String*)	
	ToString()	String
Exception	*Properties*	
	Context	String, RO
	DefaultText	String
	MessageNumber	Number, RO
	MessageSetNumber	Number, RO
	MessageSeverity	String, RO
	SubstitutionCount	Number, RO
Field	*Function*	
	GetField([*recname.fieldname*])	Field object
Field	*Methods*	
	AddDropDownItem(*CodeString*, *DescriptionString*)	
	ClearDropDownList()	
	GetAuxFlag(*FlagNumber*)	Boolean
	GetLongLabel(*LabelID*)	String

Class Name	Functions, Methods and Properties	Returns
	GetRelated(*recname.fieldname*)	Field object
	GetShortLabel(*LabelID*)	String
	SearchClear()	
	SetCursorPos(PAGE.*pagename* | %Page)	
	SetDefault()	
Field	*Properties*	
	DataAreaCollapsed	Boolean
	DisplayFormat	String
	DisplayOnly	Boolean
	EditError	Boolean
	Enabled	Boolean
	FieldLength	Number, RO
	FormatLength	Number, RO
	FormattedValue	
	IsAltKey	Boolean, RO
	IsAuditFieldAdd	Boolean, RO
	IsAuditFieldChg	Boolean, RO
	IsAuditFieldDel	Boolean, RO
	IsAutoUpdate	Boolean, RO

Class Name	Functions, Methods and Properties	Returns
	IsChanged	Boolean, RO
	IsDateRangeEdit	Boolean, RO
	IsDescKey	Boolean, RO
	IsDuplKey	Boolean, RO
	IsEditTable	Boolean, RO
	IsEditXlat	Boolean, RO
	IsFromSearchField	Boolean, RO
	IsInBuf	Boolean, RO
	IsKey	Boolean, RO
	IsListItem	Boolean, RO
	IsNotUsed	Boolean, RO
	IsRequired	Boolean, RO
	IsSearchItem	Boolean, RO
	IsSystem	Boolean, RO
	IsThroughSearchField	Boolean, RO
	IsUseDefaultLabel	Boolean, RO
	IsYesNo	Boolean, RO
	Label	String
	LabelImage	{Image.*imagename* | String}

Class Name	Functions, Methods and Properties	Returns
	LongTranslateValue	Any
	MessageNumber	Number, RO
	MessageSetNumber	Number, RO
	Name	String, RO
	OriginalValue	Depends on field
	ParentRecord	Record Object, RO
	PromptTableName	String, RO
	SearchDefault	Boolean
	SearchEdit	Boolean
	ShortTranslateValue	Any
	ShowRequiredFieldCue	Boolean
	SqlText	String
	StoredFormat	String, RO
	Style	String
	Type	String, RO
	Value	Depends on field
	Visible	Boolean
File	*Functions*	
	CreateDirectory(*path*, [, *pathtype*])	

Class Name	Functions, Methods and Properties	Returns
	FileExists(*filename* [, *pathtype*])	Boolean
	FindFiles(*filespec_pattern* [, *pathtype*])	Array object
	GetFile(*filename*, *mode* [, *charset*] [,*pathtype*])	File object
File	*Methods*	
	Close()	
	CreateRowset()	Rowset object
	GetPosition()	Number
	Open(*filename*, *mode* [, *charset*] [, *pathtype*])	
	ReadLine(*string*)	Boolean
	ReadRowset()	Rowset object
	SetFileId(*fileid*, *position*)	
	SetFileLayout (FILELAYOUT.*filelayoutname*)	Boolean
	SetPosition(*position*)	
	SetRecTerminator(*Terminator*)	
	WriteLine(*string*)	
	WriteRaw(*RawBinary*)	
	WriteRecord(*record*)	Boolean
	WriteRowset(*rowset* [, *Write_Data*])	Boolean

Class Name	Functions, Methods and Properties	Returns
	WriteString(*string*)	
File	*Properties*	
	CurrentRecord	String, RO
	IgnoreInvalidId	Boolean
	IsError	Boolean, RO
	IsNewFileId	Boolean, RO
	IsOpen	Boolean, RO
	Name	String, RO
Grid	*Function*	
	GetGrid(PAGE.*pagename*, *gridname* [, *occursnumber*])	Grid object
Grid	*Methods*	
	*GetColumn(*columnname)*	Grid column object
	Properties	
	gridcolumn	Grid column object
	Label	String
GridColumn	*Properties*	
	Enabled	Boolean
	Label	String
	Name	String, RO

Class Name	Functions, Methods and Properties	Returns
	Visible	Boolean
Internet Script	See Appendix A, Appendix A: Cheat Sheets , Internet Script Classes Methods and Properties , page 1924.	
Java	*Functions*	
	CreateJavaArray (*ElementClassName[], NumberOfElements*)	Java object
	CreateJavaObject(*ClassName* [*ConstructorParams*])	Java Object
	GetJavaClass(*ClassName*)	Java Object
Message	*Functions*	
	AddSystemPauseTimes(*StartDay*, *StartTime*, *EndDay*, *EndTime*)	Boolean
	CreateMessage (MESSAGE.*messagename*)	Message object
	DeleteSystemPauseTimes(*StartDay*, *StartTime*, *EndDay*, *EndTime*)	Boolean
	GetMessage()	Message object
	GetSyncLogData(*GUID*, *pubnode*, *chnlname*, *msg_name*, *logtype* [, *message_version*])	String
	IsMessageActive(*Message. Message_Name*)	Boolean
	NodeDelete(*nodeName*)	Boolean
	NodeRename(*oldNodeName*, *newNodeName*)	Boolean

Class Name	Functions, Methods and Properties	Returns
	NodeSaveAs(*oldNodeName*, *newNodeName*)	Boolean
	PingNode(*MsgNodeName*)	Array of number
	ReturnToServer({TRUE | FALSE | *&NODE_ARRAY*})	
	SetChannelStatus(*ChannelName*, *Status*)	Boolean
Message	*Methods*	
	Clone()	Message object
	CopyRowset(*source_rowset*, [, *record_list*])	
	CopyRowsetDelta(*source_rowset*, [, *record_list*])	
	CopyRowsetDeltaOriginal(*source_rowset*, [, *record_list*])	
	ExecuteEdits([*editlevels*])	
	GenXMLString()	String
	GetQueryString(*Parameter_Name*)	String
	GetRowset()	Rowset object
	LoadXMLString(*XMLString*)	
	Publish([*Deferred_Mode*] [, *NodeName*])	Boolean
	SetEditTable(*%PromptTable*, RECORD.*recname*)	

Class Name	Functions, Methods and Properties	Returns
	SetStatus(*status*)	
	SyncRequest([*NodeName*])	message object
	Update([*versionlist*])	
Message	*Properties*	
	ChannelName	String, RO
	Cookies	String
	DoNotPubToNodeName	String
	GUID	String, RO
	IsActive	Boolean, RO
	IsDelta	Boolean, RO
	IsEditError	Boolean, RO
	IsLocal	Boolean, RO
	MessageDetail	String
	Name	String, RO
	NRId	String, RO
	PubID	Number, RO
	PubNodeName	String, RO
	ResponseStatus	Number, RO
	Size	Number, RO

Class Name	Functions, Methods and Properties	Returns
	SubName	String, RO
	SubscriptionProcessID	String, RO
Mobile	See Appendix A, Appendix A: Cheat Sheets , Mobile Classes Methods and Properties , page 1920.	
MultiChannel Framework	*Functions*	
	DeQueue(*physical queue ID, task type, task number, agent ID*)	Number
	EnQueue(*logical queue , task type, Relative URL, Language_Code* [,*subject*][, *agent ID*][, *overflow timeout*][, *escalation timeout*][, *cost*][, *priority*][, *skill level*])	Number
	Forward(*from physical queue ID, from agent ID, task number, task type, to logical queue ID*[, *to agent ID*])	Number
	InitChat(*logical queue ID, application data URL, customer username,* [*chat_subject*][, *chat_question*][, *wizard_URL*][, *priority*][, *skill_level*])	Number
	NotifyQ(*logical queue ID, task type*)	Number
MultiChannel Framework — MCFGetMail	*Methods*	
	ReadEmailWithAttach(*User, password, server, node, UID*)	Rowset object
	ReadEmailsWithAttach(*User, password, server, node, count*)	Rowset object
	ReadAllEmailHeadersWithAttach (*user, password, server, node*)	Rowset object

Class Name	Functions, Methods and Properties	Returns
	GetEmailCount(*user*, *password*, *server*, *node*)	Number
	RemoveEmail(*user*, *password*, *server*, *node*, *UID list*)	Boolean
MultiChannel Framework — MCFMailStore	*Methods*	
	AuthorizeEmailAttach(*email ID*, *authentication name*, *authentication type* , *operation*)	Boolean
	DeleteEmail*(email ID*, [*forced delete*])	Boolean
	RetrieveEmail(*email ID*)	Rowset object
	StoreEmail(*rowset*, *row*)	Number
Notification	*Constructors*	
	Notification(*NotifyFrom*, *dttmCreated*, *language_cd*)	Notifiacation object
	NotificationAddress(*Oprid*, *Description*, *Language*, *EmailId*, *Channel*)	
	NotificationTemplate(*ComponentId*, *Market*, *TemplateId*, *TemplateType*)	NotificationTemplate object
	WorklistEntry()	WorklistEntry object
Notification	*Methods*	
	Send()	
Notification	*Properties*	
	ContentType	String

Class Name	Functions, Methods and Properties	Returns
	dttmCreated	Datetime
	EmailReplyTo	String
	FileNames	Array of String
	FileTitles	Array of String
	language_cd	String
	Message	String
	NotifyBCC	Array of NotificationAddress objects
	NotifyCC	Array of NotificationAddress objects
	NotifyFrom	String
	NotifyGuid	GUID
	NotifyTo	Array of NotificationAddress objects
	SourceComponent	String
	SourceMarket	String
	SourceMenu	String
	Subject	String
	Template	String
NotificationAddress	*Properties*	
	Channel	String

Class Name	Functions, Methods and Properties	Returns
	Description	String
	Emailld	String
	Language	String
	Oprid	String
NotificationTemplate	*Methods*	
	GetAndExpandTemplate(*Language, Vars*)	Boolean
	SetupCompVarsAndRcpts (*&Rowset_Context*)	String
	SetupGenericVars(*&AryValues*)	String
NotificationTemplate	*Properties*	
	ComponentId	String
	Instruction	String
	Language	String
	Market	String
	Priority	String
	Responses	String
	Subject	String
	TemplateId	String
	TemplateType	String

Class Name	Functions, Methods and Properties	Returns
	Text	String
OptEngine	*Functions*	
	CreateOptEngine(*probinst*, {%Synch | %ASynch} [, *Machine_Name*])	OptEngine object
	DeleteOptProbInst(*probinst*)	Integer
	GetOptEngine(*probinst*)	Integer
	GetOptProbInstList(*ProblemType* , *OutputErrorCode* [, *Prefix*])	Array of string
	InsertOptProbInst(*probinst*, *ProblemType*)	Number
OptEngine	*Methods*	
	CheckOptEngineStatus()	Number
	FillRowset(*PARAM_NAME*, *&Rowset*)	Boolean
	GetDate(*PARAM_NAME*)	Date
	GetDateArray(*PARAM_NAME*)	Array of Date
	GetDateTime(*PARAM_NAME*)	DateTime
	GetDateTimeArray(*PARAM_ NAME*)	Array of Datetime
	GetNumber(*PARAM_NAME*)	Number
	GetNumberArray(*PARAM_NAME*)	Array of number
	GetString(*PARAM_NAME*)	String

Class Name	Functions, Methods and Properties	Returns
	GetStringArray(*PARAM_NAME*)	Array of string
	GetTime(*PARAM_NAME*)	Time
	GetTimeArray(*PARAM_NAME*)	Array of Time
	RunAsynch(*TRANSACTION, PARM_PAIRS*)	Boolean
	RunSynch(*TRANSACTION, PARM_PAIRS*)	Boolean
	ShutDown()	Boolean
OptEngine	*Property*	
	DetailMsgs	Array of String, RO
Page	*Function*	
	GetPage(PAGE.*pagename*)	Page object
Page	*Properties*	
	DisplayOnly	Boolean, RO
	Name	String, RO
	Visible	Boolean
PortalRegistry	See Appendix A, Appendix A: Cheat Sheets , PortalRegistry Classes Methods and Properties , page 1873.	
PostReport	*Function*	
	SetPostReport()	PostReport object
PostReport	*Methods*	

Class Name	Functions, Methods and Properties	Returns
	AddDistributionOption(*DistIdType, DistId*)	Number
	Put()	
PostReport	*Properties*	
	ExpirationDate	Date
	OutDestFormat	String
	ProcessInstance	Number
	ProcessName	String
	ProcessType	String
	ReportDesc	String
	ReportFolder	String
	ReportId	Number
	ServerName	String
ProcessRequest	*Function*	
	CreateProcessRequest([*ProcessType, ProcessName*])	ProcessRequest object
	GetNextProcessInstance([*Commit*])	Number
	SetupScheduleDefnItem (*ScheduleName, JobName*)	ProcessRequest object
ProcessRequest	*Method*	

Class Name	Functions, Methods and Properties	Returns
	AddDistributionOption(*DistIdType*, *DistId* [, *JobName*] [, *PrcsItemLevel*] [, *JobSeqNo*])	Number
	PrintJobHTMLRpt()	String
	PrintJobRqstRpt(*ProcessInstance*, *ItemInstance* [, *PrintJobTree*] [, *PrintDistList*] [, *PrintNotifyList*] [, *PrintSystemMessage*] [, *PrintApplicationMessage*] [, *PrintParamList*])	String
	PrintSchdlHTMLRpt([*PrintJobTree*] [, *PrintDistList*] [, *PrintNotifyList*] [, *PrintMessageList*] [, *PrintParamList*])	String
	RunJobSetNow()	
	Schedule()	
	SetEmailOption(*EmailSubject*, *EmailText*, *EmailAddress*, *EmailWebReport*, *EmailAttachLog* [, *JobName*] [, *PrcsItemLevel*] [, *JobSeqNo*])	
	SetItemFolder(*PortalFolder* [, *JobName*] [, *PrcsItemLevel*] [, *JobSeqNo*])	
	SetOutputOption(*OutputType*, *OutputFormat*, *OutputDest* [, *JobName*][, *PrcsItemLevel*] [, *JobSeqNo*])	
	UpdateRunStatus()	
ProcessRequest	*Properties*	
	EmailAttachLog	Boolean

Class Name	Functions, Methods and Properties	Returns
	EmailSubject	String
	EmailText	String
	EmailWebReport	Boolean
	JobName	String
	LanguageCd	String
	NotifyTextMsgNum	Number
	NotifyTextMsgSet	Number
	OutDest	String
	OutDestFormat	String
	OutDestType	String
	PortalFolder	String
	ProcessInstance	Number
	ProcessName	String
	ProcessType	String
	RunControlID	String
	RunDateTime	DateTime
	RunLocation	String
	RunRecurrence	String
	RunStatus	Number

Class Name	Functions, Methods and Properties	Returns
	Status	Number, RO
	TimeZone	String
Record	*Functions*	
	CreateRecord(RECORD.*recname*)	Record object
	GetRecord([RECORD.*recname*])	Record object
Record	*Methods*	
	CompareFields(*RecordObject*)	Boolean
	CopyChangedFieldsTo(*RecordObj*)	
	CopyFieldsTo(*RecordObject*)	
	DBPatternMatch(*Value*, *Pattern*, *CaseSensitive*)	Boolean
	Delete()	Boolean
	ExecuteEdits([*EditLevel*])	
	*GetField(*n*/FIELD.*fieldname*)	Field Object
	Insert()	Boolean
	SearchClear()	
	SelectByKey()	Boolean
	SelectByKeyEffDt(*Date*)	Boolean
	SetDefault()	

Class Name	Functions, Methods and Properties	Returns
	SetEditTable(*%PromptField*, RECORD.*recname*)	
	Update([*KeyRecord*])	Boolean
Record	*Properties*	
	FieldCount	Number, RO
	fieldname	Field Object, RO
	IsChanged	Boolean, RO
	IsDeleted	Boolean, RO
	IsEditError	Boolean, RO
	Name	String, RO
	ParentRow	Row Object, RO
	RelLangRecName	String, RO
Row	*Function*	
	GetRow()	Row object
Row	*Methods*	
	CopyTo(*rowobject*)	Row Object
	GetNextEffRow()	Row Object
	GetPriorEffRow()	Row Object
	*GetRecord({*n* | RECORD.*recname*})	Record Object

Class Name	Functions, Methods and Properties	Returns
	GetRowSet({*n* | SCROLL.*scrollname*})	Rowset Object
	scrollname(*n*)	Row Object
Row	*Properties*	
	ChildCount	Number, RO
	DeleteEnabled	Boolean
	IsChanged	Boolean, RO
	IsDeleted	Boolean, RO
	IsEditError	Boolean, RO
	IsNew	Boolean, RO
	ParentRowSet	Rowset object, RO
	recname	Record object, RO
	RecordCount	Number, RO
	RowNumber	Number, RO
	Selected	Boolean
	Style	String
	Visible	Boolean
Rowset	*Functions*	
	CreateRowset({RECORD.*recname* | *&Rowset*} [, {FIELD.*fieldname*, RECORD.*recname* | *&Rowset*}] . . .)	Rowset object

Class Name	Functions, Methods and Properties	Returns
	GetLevel0()	Rowset object
	GetRowset([SCROLL.*scrollname*])	Rowset object
Rowset	*Methods*	
	CopyTo(*&DestRowset* [, RECORD.*SourceRecname*, RECORD.*DestRecname*]. . .)	
	DeleteRow(*n*)	Boolean
	Fill([*wherestring* [, *bindvalue*] . . .])	Number
	FillAppend([*wherestring* [, *bindvalue*] . . .])	Number
	Flush()	
	FlushRow(*n*)	
	GetCurrEffRow()	Row Object
	GetNewEffRow()	Row Object
	*GetRow(*n*)	Row Object
	HideAllRows()	
	InsertRow(*n*)	Boolean
	Refresh()	
	Select([*parmlist*], RECORD.*selrecord* [, *wherestr*, *bindvars*])	Number
	SelectNew([*parmlist*], RECORD.*selrecord* [, *wherestr*, *bindvars*])	Number

Class Name	Functions, Methods and Properties	Returns
	SetDefault(*recname.fieldname*)	
	ShowAllRows()	
	Sort([*paramlist,*] *sort_fields*)	
Rowset	*Properties*	
	ActiveRowCount	Number, RO
	ChangeOnInit	Boolean
	DataAreaCollapsed	Boolean
	DBRecordName	String, RO
	DeleteEnabled	Boolean
	EffDt	Date, RO
	EffSeq	Number, RO
	InsertEnabled	Boolean
	IsEditError	Boolean, RO
	Level	Number, RO
	Name	String, RO
	ParentRow	Row Object, RO
	ParentRowSet	Rowset Object, RO
	RowCount	Number, RO
	TopRowNumber	Number, RO

Class Name	Functions, Methods and Properties	Returns
Session	See Appendix A, Appendix A: Cheat Sheets , Session Class Methods and Properties , page 1859.	
Session	*Function*	
	GetSession()	Session object
Session	*Methods*	
	FindCompIntfcs(*partial_name*)	Component Interface collection
	FindPortalRegistries(*partial_name*)	PortalRegistry collection
	FindQueryDBRecords()	QueryDBRecord collection
	FindQueries()	Query collection
	FindQueriesDateRange (*StartDateString*, *EndDateString*)	Query collection
	FindTree(*SetId*, *UserKey*, *TreeName*, *EffDt*, *BranchName)*	Tree collection
	FindTreeStructure(*StructureID*)	Tree structure collection
	GetCompIntfc([CompIntfc.]*name*)	Component Interface object
	GetLocalNode()	Node object
	GetNodes()	Node collection
	GetPortalRegistry()	PortalRegistry object
	GetQuery()	Query object
	GetQuerySecurityProfile()	QuerySecurityProfile object
	GetRemoteNodes()	Node collection

Class Name	Functions, Methods and Properties	Returns
	GetSearchIndexes()	SearchIndexes collection
	GetSearchQuery()	SearchQuery object
	GetTree(*SetId*, *UserKey*, *TreeName*, *EffDt*, *BranchName)*	Tree object
	GetTreeStructure(*StructureID*)	Tree structure
	SearchQueryDBRecord (*SearchType*, *Pattern*, *CaseSensitive*):	QueryDBRecord collection
	SearchPrivateQueries(*QueryType*, *SearchType*, *Pattern*, *CaseSensitive*)	Query collection
	SearchPublicQueries(*QueryType*, *SearchType*, *Pattern*, *CaseSensitive*)	Query collection
Session	*Properties*	
	ErrorPending	Boolean, RO
	PSMessages	PSMessage collection object
	PSMessageMode	Number
	RegionalSettings	Regional settings object
	Repository	Repository object
	SuspendFormating	Boolean
	TraceSettings	Trace settings object
	WarningPending	Boolean, RO
SOAPDoc	*Functions*	

Class Name	Functions, Methods and Properties	Returns
	CreateSOAPDoc()	SOAPDoc object
SOAPDoc	*Methods*	
	AddBody()	
	AddEnvelope(*SOAP_Schema*)	
	AddFault(*Fault_Code*, *Fault_String*)	
	AddHeader()	
	AddMethod(*MethodName*, *IsRequest*)	
	AddParm(*ParmName*, *ParmValue*)	
	GetHeaderNode()	XmlDoc node
	GetParmName(*Index*)	String
	GetParmValue(*Index*)	String
	ValidateSOAPDoc()	String
SOAPDoc	*Properties*	
	BodyNode	XmlNode object, RO
	EnvelopeNode	XmlNode object, RO
	FaultCode	Number, RO
	FaultString	String, RO
	MethodName	String, RO

Class Name	Functions, Methods and Properties	Returns
	MethodNode	XmlNode object, RO
	ParmCount	Number, RO
	XmlDoc	XmlDoc object
SQL	*Functions*	
	CreateSQL(*sqlstring* [, *paramlist*])	
	DeleteSQL([SQL.]*sqlname* [, *dbtype* [, *effdt*]]])	Boolean
	FetchSQL([SQL.]*sqlname* [, *dbtype* [, *effdt*]])	String
	GetSQL(SQL.*sqlname* [, *paramlist*])	SQL object
	StoreSQL(*sqlstring*, [SQL.]*sqlname* [, *dbtype* [, *effdt*]])	
SQL	*Methods*	
	Close()	Boolean
	Execute(*paramlist*)	Boolean
	Fetch(*paramlist*)	Boolean
	Open(*sql* [, *paramlist*])	
SQL	*Properties*	
	BulkMode	Boolean
	IsOpen	Boolean, RO
	LTrim	Boolean

Class Name	Functions, Methods and Properties	Returns
	ReuseCursor	Boolean
	RowsAffected	Number, RO
	Status	Number, RO
	TraceName	String
	Value	String, RO
SyncServer	*System Variable*	
	%SyncServer	SyncServer object
SyncServer	*Methods*	
	AddReference(*SyncID*[, *CIDefnName*])	
	AddReferenceType(*CIDefnName*)	
SyncServer	*Properties*	
	ClientPlatform	String
	ConflictAlgorithm	String
	ConflictStatus	String
	ExcludeProperty	String
	ExportLocation	String
	PropertyValue	String
	SessionID	String, RO
	Synchronizing	String

Class Name	Functions, Methods and Properties	Returns
	SyncIDs	Array of number
	TypeID	String
	validateID	String
	validateRowCount	Number, RO
	validateVersion	String
TransformData	*System Variable*	
	%TransformData	TransformData object
TransformData	*Properties*	
	DestMsgName	String, RO
	DestMsgVersion	String, RO
	DestMsgVersion	String, RO
	DestNode	String, RO
	SourceMsgName	String, RO
	SourceMsgVersion	String, RO
	SourceNode	String, RO
	Status	Number
	XmlDoc	XmlDoc object
Tree	See Appendix A, Appendix A: Cheat Sheets , Tree Classes Methods and Properties , page 1910.	

Class Name	Functions, Methods and Properties	Returns
WorklistEntry	*Methods*	
	Create()	Number
	Save()	Number
	Update()	Number
WorklistEntry	*Properties*	
	actiondttm	Datetime
	busactivity	String
	buseventname	String
	busprocname	String
	commentshort	String
	do_replicate_flag	String
	instanceid	Number
	instselectddttm	Datetime
	inststatus	String
	insttimeoutddttm	Datetime
	instworkeddttm	Datetime
	oprid	String
	originatorid	String
	prevoprid	String

Class Name	Functions, Methods and Properties	Returns
	syncid	Number
	timedout	String
	transactionid	Number
	url	String
	wl_priority	String
	wldaystoselect	Number
	wldaystowork	Number
	worklistname	String
XmlDoc	*Functions*	
	CancelPubHeaderXmlDoc(*PubID, PubNode, ChannelName, VersionName*)	Boolean
	CancelPubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName, SubNode*)	Boolean
	CancelSubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName, SubscriptionName*)	Boolean
	CreateXmlDoc(*XmlString*)	XmlDoc object
	GetMessageXmlDoc()	XmlDoc object
	GetNRXmlDoc(*NRID, NodeName*)	XmlDoc object

Class Name	Functions, Methods and Properties	Returns
	GetPubHeaderXmlDoc(*PubID*, *PubNode*, *ChannelName*, *VersionName*)	XmlDoc object
	GetPubXmlDoc(*PubID* ,*PubNode*, *ChannelName* ,*VersionName* ,*MessageName*,*SubNode*)	XmlDoc object
	GetSubXmlDoc(*PubID*, *PubNode*, *ChannelName*, *VersionName*, *MessageName*)	XmlDoc object
	PublishXmlDoc(*&XmlDoc*, **Message.** *MessageName* [, **Node.** *NodeName*])	Boolean
	ReSubmitPubHeaderXmlDoc (*PubID*, *PubNode*, *ChannelName*, *VersionName*)	Boolean
	ReSubmitPubXmlDoc(*PubID*, *PubNode*, *ChannelName* ,*VersionName*,*MessageName*, *SubNode*)	Boolean
	ReSubmitSubXmlDoc(*PubID* , *PubNode* , *ChannelName* , *VersionNam*,,*MessageName* , *SubscriptionName*)	Boolean
	RevalidateNRXmlDoc(*NRID*, *EntityName*	Boolean
	SyncRequestXmlDoc(*&XmlDoc*, **Message.** *MessageName* [, **Node.** *NodeName*])	XmlDoc object
	Transform(*Input*, *AE_Program_ Name*, *Initial_Node_Name*, *Initial_Message__Name*, *Initial_Message_Version*, *Result_Node_Name*, *Result_Message_Name*, *Result_Message_Version*)	XmlDoc object

Class Name	Functions, Methods and Properties	Returns
	TransformEx(*XmlString*, *XsltString*)	String
	UpdateXmlDoc(*&XmlDoc*, *PubID*, *PubNode*, *ChannelName*, *VersionName* [, *Message Name* [, *SubNode*]])	Boolean
XmlDoc	*Methods*	
	CopyRowset(*&InRowset*[, *MessageName*] [, *MessageVersion*])	Boolean
	CopyToPSFTMessage(*targetDoc*, *srcPath*, *targetPath*)	Number
	CopyToRowset(*&InRowset*[, *Message_Name*][, *Message_Version*])	Boolean
	CreateDocumentElement(*TagName* [, *NamespaceURI*] [, *&DocType*])	XmlNode object
	CreateDocumentType(*Name*, *PublicID*, *SystemID*)	String
	GenFormattedXmlString()	String
	GenXmlString()	String
	GenFormattedXmlString()	String
	GetElementsByTagName (*TagName*)	Array of XmlNode objects
	LoadIBContent(*Content*[, *RootTagName*])	Boolean
	ParseXmlFromURL(*path*)	Boolean
	ParseXmlString(*XmlString*)	Boolean

Class Name	Functions, Methods and Properties	Returns
XmlDoc	*Properties*	
	DocumentElement	XmlNode object, RO
	IsNull	Boolean, RO
XmlNode	*Methods*	
	AddAttribute(*Name, Value*)	
	AddAttributeNS(*NamespaceURI, AttributeName, Value*)	
	AddCDataSection(*Data*)	XmlNode object
	AddComment(*Text*)	XmlNode object
	AddElement(*TagName*)	XmlNode object
	AddElementNS(*NameSpaceURI, TagName*)	XmlNode object
	AddEntityReference(*Name*)	XmlNode object
	AddNode(*XmlNode*)	XmlNode object
	AddProcessInstruction(*Target, Data*)	XmlNode object
	AddText(*Data*)	XmlNode object
	CopyNode(*Node*)	
	FindNode(*Path*)	XmlNode object
	FindNodes(*Path*)	Array of XmlNode objects
	GenXmlString()	String

Class Name	Functions, Methods and Properties	Returns
	GetAttributeName(*index*)	String
	GetAttributeValue({*Name* | *Index*})	String
	GetCDataValue()	String
	GetCDataValues()	Array of string
	GetChildNode(*index*)	XmlNode object
	GetElement()	XmlNode object
	GetElements()	Array of XmlNode objects
	GetElementsByTagName (*TagName*)	Array of XmlNode objects
	GetElementsByTagNameNS (*NamespaceURI*, *TagName*)	Array of XmlNode objects
	InsertCDataSection(*Data*, *Position*)	XmlNode object
	InsertComment(*Data*, *Position*)	XmlNode object
	InsertElement(*TagName*, *Position*)	XmlNode object
	InsertElementNS(*NameSpaceURI*, *TagName*, *Position*)	XmlNode object
	InsertEntityReference(*Name*, *Position*)	XmlNode object
	InsertNode(*&NewNode*, {*&RefNode* | *Position*})	XmlNode object
	InsertProcessInstruction(*Target*, *Data*, *Position*)	XmlNode object
	InsertText(*Data*, *Position*)	XmlNode object

Class Name	Functions, Methods and Properties	Returns
	RemoveAllChildNode()	
	RemoveChildNode({*Position* | *Node*})	XmlNode object
XmlNode	*Properties*	
	AttributesCount	Number, RO
	ChildNodeCount	Number, RO
	Index	Number, RO
	IsNull	Boolean, RO
	LocalName	String, RO
	NamespaceURI	String, RO
	NextSibling	XmlNode object, RO
	NodeName	String
	NodePath	String, RO
	NodeType	String, RO
	NodeValue	String
	ParentNode	XmlNode object, RO
	Prefix	String, RO
	PreviousSibling	XmlNode object, RO

Session Class Methods and Properties

The session object controls access to the PeopleSoft APIs, and the environment in which they run. This table contains the properties and methods for controlling the environment. The API objects (Query classes, trees, and so on) are listed, but have their own separate tables.

Objects	Methods and Properties	Returns
Component Interfaces and CI Collection	See Appendix A, Appendix A: Cheat Sheets , Component Interface Class Methods and Properties , page 1862.	
PortalRegisty classes	See Appendix A, Appendix A: Cheat Sheets , PortalRegistry Classes Methods and Properties , page 1873.	
Query classes	See Appendix A, Appendix A: Cheat Sheets , Query Classes Methods and Properties , page 1893.	
PSMessage Collection	*Methods*	
	DeleteAll()	Boolean
	DeleteItem(*number*)	Boolean
	First()	PSMessage object
	Item(*number*)	PSMessage object
	Next()	PSMessage object
PSMessage Collection	*Property*	
	Count	Number, RO
PSMessage	*Properties*	
	ExplainText	String, RO
	MessageNumber	Number, RO

Objects	Methods and Properties	Returns
	MessageSetNumber	Number, RO
	Source	String, RO
	Text	String, RO
Regional Settings	*Properties*	
	ClientTimeZone	String
	CurrencyFormat	Number
	CurrencySymbol	String
	DateFormat	Number
	DateSeparator	String
	DecimalSymbol	String
	DigitGroupingSymbol	String
	LanguageCD	String
	UseLocalTime	Boolean
	1159Separator	String
	2359Separator	String
Repository	See PeopleSoft Integration Tools and Utilities, The PeopleSoft API Repository, Summary of Repository Methods and Properties.	
TraceSettings	*Properties*	
	API	Boolean

Objects	Methods and Properties	Returns
	COBOLStmtTimings	Boolean
	ConDisRollbackCommit	Boolean
	DBSpecificCalls	Boolean
	ManagerInfo	Boolean
	NetworkServices	Boolean
	NonSSBs	Boolean
	OutputUNICODE	Boolean
	PCExtFcnCalls	Boolean
	PCFcnReturnValues	Boolean
	PCFetchedValues	Boolean
	PCIntFcnCalls	Boolean
	PCListProgram	Boolean
	PCParameterValues	Boolean
	PCProgramStatements	Boolean
	PCStack	Boolean
	PCStartOfPrograms	Boolean
	PCTraceProgram	Boolean
	PCVariableAssignments	Boolean
	RowFetch	Boolean

Objects	Methods and Properties	Returns
	SQLStatement	Boolean
	SQLStatementVariables	Boolean
	SSBs	Boolean
	SybBindInfo	Boolean
	SybFetchInfo	Boolean
	TraceFile	String
Tree, Tree Collection, Tree Structure, Tree Structure Collection	See Appendix A, Appendix A: Cheat Sheets , Tree Classes Methods and Properties , page 1910.	

Component Interface Class Methods and Properties

Component Interface collections and Component Interface objects are instantiated from a session object. CompIntfPropInfo and Data Collections are instantiated from a Component Interface object.

Class Name	Functions, Methods and Properties	Returns
Component Interface Collection	*Methods*	
	First()	Component Interface structure (no data)
	Item(*number*)	Component Interface structure (no data)
	Next()	Component Interface structure (no data)
Component Interface Collection	*Properties*	
	Count	Number, RO
Component Interface	*Methods*	

Class Name	Functions, Methods and Properties	Returns
	Cancel()	Boolean
	CopyRowset(*&rowset* [,*IntialRow*] [, *record_list*])	
	CopyRowsetDelta(*&rowsetI*, [,*IntialRow*] [, *record_list*])	
	CopySetupRowset(*&rowset* [,*IntialRow*] [, *record_list*])	
	CopySetupRowsetDelta(*&rowsetI*, [,*IntialRow*] [, *record_list*])	
	Create()	Component Interface with data
	Find()	Component Interface collection
	Get()	Component Interface with data
	GetPropertyByName(*string*)	Depends on property
	Save()	Boolean
	SetPropertyByName(*string, value*)	
Component Interface	*Properties*	
	ComponentName	String, RO
	CreateKeyInfoCollection	Component InterfacePropertyInfo collection object, RO
	FindKeyInfoCollection	Component InterfacePropertyInfo collection object, RO
	GetDummyRows	Boolean
	GetHistoryItems	Boolean

Class Name	Functions, Methods and Properties	Returns
	GetKeyInfoCollection	Component InterfacePropertyInfo collection object, RO
	InteractiveMode	Boolean
	PropertyInfoCollection	Component InterfacePropertyInfo collection object, RO
	StopOnFirstError	Boolean
	Every user-defined property for an internet component *definition* can be used like a property on the instantiated *object*.	Depends on property
CompIntfPropInfo Collection	*Methods*	
	First()	Component InterfacePropertyInfo object
	Item(*number*)	Component InterfacePropertyInfo object
	Next()	Component InterfacePropertyInfo object
CompIntfPropInfo Collection	*Properties*	
	Count	Number, RO
CompIntfPropInfo	*Properties*	
	Format	String, RO
	IsCollection	Boolean, RO
	IsReadOnly	Boolean, RO
	Key	Boolean, RO

Class Name	Functions, Methods and Properties	Returns
	LabelLong	String, RO
	LabelShort	String, RO
	Length	Number, RO
	Name	String, RO
	Prompt	Boolean, RO
	PropertyInfoCollection	Component InterfacePropertyInfo collection object
	Required	Boolean, RO
	Type	String, RO
	Xlat	Boolean, RO
	YesNo	Boolean, RO
Data Collection	*Methods*	
	CurrentItem()	data collection item
	DeleteItem(*number*)	
	GetEffectiveItem(*DateString*, *SeqNo*)	data collection item
	GetEffectiveItemNum(*DateString*, *SeqNo*)	Number
	InsertItem(*number*)	data collection item
	Item(*number*)	data collection item
	ItemByKeys(*key_values*)	data collection item

Class Name	Functions, Methods and Properties	Returns
Data Collection	*Properties*	
	Count	Number, RO
	CurrentItemNumber	Number, RO

Search Classes Methods and Properties

Search Class objects are instantiated from session and PortalRegisty class methods.

Class Name	Methods and Properties	Returns
SearchQuery	*Methods*	
	Execute(*Start*, *Size*)	SearchResult collection
	Parse()	ParseResult collection
	Properties	
	HitCount	Number, RO
	IndexName	String
	KnowledgeBase	String
	Language	String
	ProcessedCount	Number, RO
	QueryText	String
ParseResult Collection	*Methods*	
	First()	ParseResult object
	Next()	ParseResult object or Null

Class Name	Methods and Properties	Returns
ParseResult Collection	*Properties*	
	ErrorCount	Number, RO
	WarningCount	Number, RO
ParseResult Class	*Properties*	
	Message	String, RO
	Severity	String, RO
SearchResult Collection	*Methods*	
	First()	SearchResult
	Item(*number*)	SearchResult
	Next()	SearchResult
SearchResult Collection	*Properties*	
	Count	Number, RO
SearchResult	*Properties*	
	Key	String, RO
	Score	Number, RO
	SearchFields	SearchFields collection, RO
SearchField Collection	*Methods*	
	First()	SearchField
	ItemByName(*Name*)	SearchField

Class Name	Methods and Properties	Returns
	Next()	SearchField
SearchField Collection	*Properties*	
	Count	Number, RO
SearchField Class	*Properties*	
	Name	String, RO
	Value	String, RO
SearchIndexes Collection	*Methods*	
	First()	SearchIndex object
	ItemByName(*Name*)	SearchIndex object
	Next()	SearchIndex object
SearchIndexes Collection	*Property*	
	Count	Number, RO
SearchIndex Class	*Method*	
	Save()	Boolean
SearchIndex Class	*Properties*	
	ExtraOptions	Any
	FSOpts	Search FS Options object, RO
	HTTPOpts	Search HTTP Options object, RO
	Languages	Search Language collection, RO

Class Name	Methods and Properties	Returns
	Location	String
	Name	String, RO
	RecOpts	SearchRecord Options collection, RO
	Schedules	Search Schedule collection, RO
	Type	String
SearchRecord Options Class	*Properties*	
	Fields	SearchRecordField collection, RO
	Filter	String
	IncrementalView	String
	RecName	String, RO
	VeggieKey	String
	ZoneOptions	String
SearchRecordField Collection	*Methods*	
	DeleteItem(*Record, Field*)	Boolean
	First()	SearchRecordField object
	InsertItem(*Record, Field*)	SearchRecordField object
	ItemByName(*Record, Field*)	SearchRecordField object
	Next()	SearchRecordField object
SearchRecordField Collection	*Property*	

Class Name	Methods and Properties	Returns
	Count	Number, RO
SearchRecordField Class	*Properties*	
	FieldName	String, RO
	IsAttachment	Boolean
	IsVerityField	Boolean
	IsWordIndex	Boolean
	RecordName	String, RO
Search Language Collection	*Methods*	
	DeleteItem(*Name*)	Boolean
	First()	Search Language object
	InsertItem(*LanguageCode*, *MapLanguageCode*)	Search Language object
	ItemByName(*LanguageCode*)	Search Language object
	Next()	Search Language object
Search Language Collection	*Property*	
	Count	
Search Language Class	*Properties*	
	LanguageCd	String, RO
	MapLanguageCd	String, RO
Search Schedule Collection	*Methods*	

Class Name	Methods and Properties	Returns
	DeleteItem(*Name*)	Boolean
	First()	Search Schedule object
	InsertItem(*RunCntrlID)*	Search Schedule object
	ItemByName(*RunCntrlID*)	Search Schedule object
	Next()	Search Schedule object
Search Schedule Collection	*Property*	
	Count	Number, RO
Search Schedule Class	*Properties*	
	BuildType	String
	RunCntrID	String, RO
	RunRecurrance	String
	ServerName	String
Search HTTP Options Class	*Properties*	
	AllowHTTPS	Boolean
	DomainLimit	String
	GlobList	String
	GlobListType	String
	LinkDepth	Number
	MIMEList	String

Class Name	Methods and Properties	Returns
	MIMEListType	String
	ProxyHost	String
	ProxyPort	Number
	StartOpts	Search Start Options collection, RO
Search FS Options Class	*Properties*	
	GlobList	String
	GlobListType	String
	MIMEList	String
	MIMEListType	String
	StartOpts	Search Start Options collection, RO
Search Start Options Collection	*Methods*	
	DeleteItem(*Value*)	Boolean
	First()	Search Start Options object
	InsertItem(*Value*)	Search Start Options object
	ItemByName(*Value*)	Search Start Options object
	Next()	Search Start Options object
Search Start Options Collection	*Property*	
	Count	
Search Start Options Class	*Properties*	

Class Name	Methods and Properties	Returns
	IsDomainRestricted	Boolean
	IsHostRestricted	Boolean
	Value	String, RO

PortalRegistry Classes Methods and Properties

All PortalRegistry classes are instantiated from a session object.

Class Name	Methods and Properties	Returns
PortalRegistry	*Methods*	
	BuildSearchIndex(*Language*)	Boolean
	Close()	Boolean
	CopyObject(*sourcePortalName*, *sourceRefType*, *sourceObjName*, *targetPortalName*, *targetPrntFldrName*, *copyChildren*)	Boolean
	Create(*RegistryName*)	Boolean
	CreateRemote(*PortalName*, *RemoteNodeName*)	Boolean
	Delete(*RegistryName*)	Boolean
	DeleteHomepage()	Boolean
	FindCRefByName(*Name*)	ContentReference object
	FindCRefByURL(*URL*)	ContentReference object
	FindFolderByName(*Name*)	Folder object
	FindPgltByName(*PageletName*)	Pagelet object

Class Name	Methods and Properties	Returns
	GetAbsoluteContentURL(*Node. NodeName, URL*)	ContentReference object
	GetDefaultHPTabOID()	String
	GetQualifiedURL(*ContentProvider, RelativeURL*)	URL string
	GetSearchQuery()	SearchQuery object
	GrantPermissionForComponent (*MenuName, ComponentName, Market, PermListName, NodeName*)	Boolean
	GrantPermissionForScript (*RecordName, FieldName, EventName, FuncName, PermListName, NodeName*)	Boolean
	Open(*RegistryName*)	Boolean
	PermissionListDelete (*PermListName*)	Boolean
	PermissionListSaveAs (*PermListSourceName, PermListTargetName*)	Boolean
	RevokePermissionForComponent (*MenuName, ComponentName, Market, PermListName, NodeName*)	Boolean
	RevokePermissionForScript (*RecordName, FieldName, EventName, FuncName, PermListName, NodeName*)	Boolean
	Save()	Boolean
PortalRegistry	*Properties*	
	DefaultTemplate	String

Class Name	Methods and Properties	Returns
	Description	String
	Favorites	Favorite Collection, RO
	Homepage	Homepage object, RO
	Name	String, RO
	NodeTemplates	NodeTemplate Collection, RO
	OwnerId	String
	PageletCategories	PageletCategories Collection, RO
	Portals	Portal collection, RO
	RootFolder	Folder object, RO
	TabDefinitions	TabDefinitions Collection, RO
	TemplateObject	ContentReference object, RO
PortalRegistry Collection	*Methods*	
	First()	PortalRegistry object
	Item(*number*)	PortalRegistry object
	Next()	PortalRegistry object
PortalRegistry Collection	*Property*	
	Count	Number, RO
Node	*Properties*	
	ActiveNode	Boolean, RO

Class Name	Methods and Properties	Returns
	AppRelease	String, RO
	ContentURI	String, RO
	DefaultPortalName	String, RO
	Description	String, RO
	Name	String, RO
	NodePassword	String, RO
	NodeType	String, RO
	PortalURI	String, RO
	ToolsRelease	String, RO
Node Collection	*Methods*	
	First()	Node object
	ItemByName(*NodeName*)	Node object
	Next()	Node object
Node Collection	*Property*	
	Count	Number, RO
Remote Node Collection	*Methods*	
	First()	Node object
	ItemByName(*NodeName*)	Node object
	Next()	Node object

Class Name	Methods and Properties	Returns
Remote Node Collection	*Property*	
	Count	Number, RO
Portal Class	*Method*	
	Save()	Boolean
Portal Class	*Properties*	
	HostNodeName	String
	IsLocal	Boolean, RO
	Name	String, RO
Portal Collection	*Methods*	
	First()	Portal object
	ItemByName(*PortalName*)	Portal object
	Next()	Portal object
Portal Collection	*Property*	
	Count	Number, RO
NodeTemplate Class	*Properties*	
	DefaultTemplate	String
	Name	String
	TemplateObject	Template object, RO
NodeTemplate Collection	*Methods*	

Class Name	Methods and Properties	Returns
	DeleteItem(*NodeName*)	Boolean
	First()	NodeTemplate object
	InsertItem(*NodeName*)	NodeTemplate object
	ItemByName(*NodeName*)	NodeTemplate object
	Next()	NodeTemplate Object
NodeTemplate Collection	*Property*	
	Count	Number, RO
Folder	*Method*	
	Save()	Boolean
Folder	*Properties*	
	Attributes	Attribute collection object, RO
	Author	String, RO
	AuthorAccess	Boolean
	Authorized	Boolean
	CascadedPermissions	PermissionValue collection RO
	ContentRefs	ContentReference Collection, RO
	CreationDate	String, RO
	Description	String
	Folders	Folder collection, RO

Class Name	Methods and Properties	Returns
	IsMobile	Boolean
	IsVisible	Boolean, RO
	Label	String
	Name	String, RO
	OwnerID	String
	ParentName	String, RO
	Path	String, RO
	Permissions	PermissionValue collection, RO
	Product	String
	SequenceNumber	Number
	PublicAccess	Boolean
	ValidFrom	String
	ValidTo	String
Folder Collection	*Methods*	
	DeleteItem(*FolderName*)	Boolean
	First()	Folder object
	InsertItem(*FolderName*, *Label*)	Folder object
	ItemByName(*FolderName*)	Folder object
	Next()	Folder object

Class Name	Methods and Properties	Returns
Folder Collection	*Properties*	
	Count	Number, RO
ContentReference	*Method*	
	Save()	Boolean
ContentReference	*Properties*	
	AbsoluteContentURL	String, RO
	AbsolutePortalURL	String, RO
	AssignedPagelets	AssignedPagelets collection, RO
	Attributes	Attribute collection object, RO
	Author	String, RO
	AuthorAccess	Boolean
	Authorized	Boolean
	CascadedPermissions	PermissionValue collection, RO
	ContentProvider	String
	CreationDate	String, RO
	Data	String
	Description	String
	HtmlText	String, RO
	IsMobile	Boolean

Class Name	Methods and Properties	Returns
	IsVisible	Boolean, RO
	Label	String
	Name	String, RO
	OwnerId	String
	ParentName	String, RO
	Path	String, RO
	Permissions	PermissionValue collection, RO
	Product	String
	PublicAccess	Boolean
	RelativeURL	String, RO
	SequenceNumber	Number
	StorageType	String
	Template	String
	TemplateObject	ContentReference object, RO
	TemplateType	String
	URL	String
	URLType	String
	UsageType	String
	ValidFrom	String

Class Name	Methods and Properties	Returns
	ValidTo	String
ContentReference Collection	*Methods*	
	DeleteItem(*ContentReferenceName*)	Boolean
	First()	ContentReference object
	InsertItem(*ContentReferenceName*, *ContentReferenceLabel, Node, URL*)	ContentReference object
	ItemByName (*ContentReferenceName*)	ContentReference object
	Next()	ContentReference object
ContentReference Collection	*Property*	
	Count	Number, RO
AttributeValue	*Properties*	
	Label	String
	Name	String, RO
	Translatable	Boolean
	Value	String
Attribute Collection	*Methods*	
	DeleteItem(*AttributeValueName*)	Boolean
	First()	AttributeValue object
	InsertItem(*AttributeValueName*)	AttributeValue object
	ItemByName(*AttributeValueName*)	AttributeValue object

Class Name	Methods and Properties	Returns
	Next()	AttributeValue object
Attribute Collection	*Property*	
	Count	Number, RO
PermissionValue	*Properties*	
	Cascade	Boolean
	Name	String
PermissionValue Collection	*Methods*	
	DeleteItem(*PermissionValueName*)	Boolean
	First()	PermissionValue object
	InsertItem(*PermissionValueName*)	PermissionValue object
	ItemByName (*PermissionValueName*)	PermissionValue object
	Next()	PermissionValue object
PermissionValue Collection	*Property*	
	Count	Number, RO
TabDefinition Class	*Method*	
	Save()	Boolean
TabDefinition Class	*Properties*	
	AssignedPagelets	AssignedPagelets collection, RO
	AvailableCategories	AvailableCategories collection, RO

Class Name	Methods and Properties	Returns
	AvailablePagelets	AvailablePagelets collection, RO
	Attributes	Attribute collection object, RO
	Author	String, RO
	AuthorAccess	Boolean
	Authorized	Boolean
	ColumnLayout	String
	CreationDate	String, RO
	Description	String
	DynamicCategories	DynamicCategories collection, RO
	HelpID	String
	HtmlText	String, RO
	IsHideActionBar	Boolean
	IsLayoutLocked	Boolean
	IsRenamable	Boolean
	Label	String
	LayoutBehavior	String
	Name	String, RO
	OwnerId	String
	Product	String

Class Name	Methods and Properties	Returns
	PublicAccess	Boolean
	QualifiedURL	String, RO
	SequenceNumber	Number
	StyleSheet	String
	ValidFrom	String
	ValidTo	String
TabDefinition Collection	*Methods*	
	DeleteItem(*TabDefinitionName*)	Boolean
	First()	TabDefinition object
	InsertItem(*TabDefinitionName*)	TabDefinition object
	ItemByName(*TabDefinitionName*)	TabDefinition object
	Next()	TabDefinition object
TabDefinition Collection	*Property*	
	Count	Number, RO
AssignedPagelet Class	*Properties*	
	Column	Number
	LayoutBehavior	String
	PageletName	String, RO
	Row	Number

Class Name	Methods and Properties	Returns
AssignedPagelet Collection	*Methods*	
	DeleteItem(*PageletName*)	Boolean
	First()	Pagelet object
	InsertItem(*PageletName*, *Column*, *Row*, *LayoutBehavior*)	Pagelet object
	ItemByName(*PageletName*)	Pagelet object
	Next()	Pagelet object
AssignedPagelet Collection	*Property*	
	Count	Number, RO
AvailableCategory Class	*Properties*	
	AvailablePagelets	AvailablePagelets Collection, RO
	CategoryName	String, RO
AvailableCategory Collection	*Methods*	
	First()	AvailableCategory object
	ItemByName(*Name*)	AvailableCategory object
	Next()	AvailableCategory object
AvailableCategory Collection	*Property*	
	Count	Number, RO
AvailablePagelet Class	*Properties*	
	CategoryLabel	String, RO

Class Name	Methods and Properties	Returns
	CategoryName	String, RO
	Column	Number, RO
	LayoutBehavior	String, RO
	PageletLabel	String, RO
	PageletName	String, RO
	Row	Number, RO
AvailablePagelet Collection	*Methods*	
	First()	AvailablePagelet object
	ItemByName(*Name*)	AvailablePagelet object
	Next()	AvailablePagelet object
AvailablePagelet Collection	*Property*	
	Count	Number, RO
DynamicCategory Collection	*Methods*	
	DeleteItem(*Name*)	Boolean
	First()	String
	InsertItem(*Name*)	String
	ItemByName(*Name*)	String
	Next()	String
DynamicCategory Collection	*Property*	

Class Name	Methods and Properties	Returns
	Count	Number, RO
PageletCategory Class	*Method*	
	Save()	Boolean
PageletCategory Class	*Properties*	
	Attributes	Attribute collection object, RO
	Author	String, RO
	AuthorAccess	Boolean
	Authorized	Boolean
	CascadedPermissions	PermissionList object, RO
	CreationDate	String, RO
	Description	String
	Label	String
	Name	String, RO
	OwnerId	String
	Pagelets	Pagelets Collection, RO
	Permissions	PermissionList Collection, RO
	Product	String
	PublicAccess	Boolean
	SequenceNumber	Number

Class Name	Methods and Properties	Returns
PageletCategory Collection	*Methods*	
	DeleteItem(*Name*)	Boolean
	First()	String
	InsertItem(*Name*, *Label*)	String
	ItemByName(*Name*)	String
	Next()	String
PageletCategory Collection	*Property*	
	Count	Number, RO
Pagelet Class	*Method*	
	Save()	Boolean
Pagelet Class	*Properties*	
	Attributes	Attribute collection object, RO
	Author	String, RO
	AuthorAccess	Boolean
	Authorized	Boolean
	CascadedPermissions	PermissionList Collection, RO
	ContentProvider	Node object, RO
	CreationDate	String, RO
	DefaultColumn	Number

Class Name	Methods and Properties	Returns
	Description	String
	HelpID	String
	IsHideMinimize	Boolean
	Label	String
	Name	String, RO
	OwnerId	String
	ParentName	String, RO
	Permissions	PermissionList Collection, RO
	Product	String
	PublicAccess	Boolean
	QualifiedURL	String, RO
	SequenceNumber	Number
	URL	String
	URLType	String
Pagelet Collection	*Methods*	
	DeleteItem(*Name*)	Boolean
	First()	Pagelet object
	InsertItem(*Name, Label, NodeName, URL*)	Pagelet object
	ItemByName(*Name*)	Pagelet object

Class Name	Methods and Properties	Returns
	Next()	Pagelet object
Pagelet Collection	*Property*	
	Count	Number, RO
UserHomepage	*Method*	
	Save()	
UserHomepage	*Properties*	
	Greeting	String
	UserId	String, RO
	UserTab	String, RO
UserTab	*Properties*	
	ColumnLayout	Number
	Label	String
	QualifiedURL	String, RO
	SelectedPagelets	SelectedPagelets, RO
	SequenceNumber	Number
	TabName	String, RO
UserTab Collection	*Methods*	
	DeleteItem(*Name*)	Boolean
	First()	UserTab object

Class Name	Methods and Properties	Returns
	InsertItem(*Name*)	UserTab object
	ItemByName(*Name*)	UserTab object
	Next()	UserTab object
UserTab Collection	*Property*	
	Count	Number, RO
SelectedPagelet Class	*Properties*	
	CategoryName	String, RO
	Column	Number
	IsMinimized	Boolean
	PageletName	String, RO
	Row	Number
SelectedPagelet Collection	*Method*	
	DeleteItem(*Name*)	Boolean
	First()	SelectedPagelet object
	InsertItem(*Name*)	SelectedPagelet object
	ItemByName(*Name*)	SelectedPagelet object
	Next()	SelectedPagelet object
SelectedPagelet Collection	*Property*	
	Count	Number, RO

Class Name	Methods and Properties	Returns
Favorites Class	*Properties*	
	CRefName	String, RO
	Label	String
	QualifiedURL	String, RO
	SequenceNumber	Number
	URL	String
Favorites Collection	*Methods*	
	DeleteItem(*FavoriteLabel*)	Boolean
	First()	Favorite object
	InsertItem(*FavoriteLabel*, *FavoriteName*)	Favorite object
	ItemByLabel(*FavoriteLabel*)	Favorite object
	Next()	Favorite object
Favorites Collection	*Property*	
	Count	Number, RO

Query Classes Methods and Properties

Query classes are instantiated from a session object.

Class Name	Methods and Properties	Returns
Query Collection	*Methods*	
	First()	Query object

Class Name	Methods and Properties	Returns
	Item(*Number*)	Query object
	ItemByName(*Name*)	Query object
	Next()	Query object
Query Collection	*Property*	
	Count	Number, RO
Query Class	*Methods*	
	AddQuerySelect()	QuerySelect object
	AddPrompt(*PromptName*)	Prompt object
	Close()	
	Create(*QueryName*, *Pubic*, *Type*, *Description*, *LongDescription*)	Query object
	Delete()	Number
	DeletePrompt(*PromptName*)	Number
	FindExpression(*Number*)	Expression object
	FormatResultString(*&Rowset*, *Output_Format*, *StartRow*, *EndRow*)	String
	Open(*QueryName*, *Public*, *Update*)	Number
	Rename(*NewQueryName*)	Number
	RunToFile(*&PromptRecord*, *Destination*, *OutputFormat*, *MaxRows*)	Number
	RunToRowset(*&PromptRecord*, *MaxRows*)	Rowset object

Class Name	Methods and Properties	Returns
	Save()	Number
Query Class	*Properties*	
	Approved	String
	ApprovedDtTm	String, RO
	ApprovedUserId	String
	CreateDtTm	String, RO
	CreateUserId	String, RO
	Description	String
	ExecAppName	String
	ExecLogging	Boolean
	LastSQLErrorCode	Number, RO
	LastUpdDttm	String, RO
	LastUpdOprId	String, RO
	LongDescription	String
	Metadata	Metadata collection, RO
	Name	String, RO
	PDFFont	String
	Prompts	QueryPrompt collection, RO
	PromptRecord	Record object, RO

Class Name	Methods and Properties	Returns
	PublicPrivate	String
	QuerySelect	QuerySelect object, RO
	QueryStatistics	QueryStatistics object, RO
	RunTimePrompts	QueryPrompt Collection, RO
	SQL	String, RO
	Type	Number
QuerySelect Class	*Methods*	
	AddAllFields(*QueryRecord*)	Number
	AddCriteria(*Name*)	QueryCriteria object
	AddExpression(*Name*)	QueryExpression object
	AddHavingCriteria(*Name*)	QueryCriteria object
	AddQueryExpressionField(*Name*)	QueryField object
	AddQueryOutputField (*QueryRecord*, *index*)	QueryField object
	AddQueryRecord (*QueryRecordName*)	QueryRecord object
	AddQuerySelectedField (*QueryRecord*, *index*)	QueryField object
	DeleteCriteria(*index*)	Number
	DeleteExpression(*Index*)	Number
	DeleteField(*Index)*	Number

Class Name	Methods and Properties	Returns
	DeleteHavingCriteria(*Index*)	Number
	DeleteRecord(*Index*)	Number
QuerySelect	*Properties*	
	Criteria	QueryCriteria collection, RO
	Distinct	Boolean
	Expressions	QueryExpression collection, RO
	HavingCriteria	QueryCriteria collection, RO
	ParentSelectNum	Number, RO
	QueryOutputFields	QueryField collection, RO
	QueryRecords	QueryRecord collection, RO
	QuerySelectedFields	QueryField collection, RO
	QuerySelects	QuerySelect object, RO
	SelectNum	Number, RO
	SelectType	Number, RO
QuerySelect Collection	*Methods*	
	AddUnion()	QuerySelect object
	DeleteQuerySelect(*SelNumber*)	Number
	First()	QuerySelect object
	Item(*Number*)	QuerySelect object

Class Name	Methods and Properties	Returns
	ItemBySelNum(*SelNumber*)	QuerySelect object
	Next()	QuerySelect object
Query Collection	*Property*	
	Count	Number, RO
QueryRecord Collection	*Methods*	
	First()	QueryRecord object
	Item(*Number*)	QueryRecord object
	ItemByAlias(*Alias*)	QueryRecord object
	Next()	QueryRecord object
QueryRecord Collection	*Property*	
	Count	Number, RO
QueryRecord	*Method*	
	GetField(*Index*)	QueryField object
QueryRecord	*Properties*	
	Description	String, RO
	JoinAlias	String
	JoinFieldName	String
	JoinType	Number
	Name	String, RO

Class Name	Methods and Properties	Returns
	QueryFields	QueryFields collection, RO
	RecordAlias	String
QueryField Collection	*Methods*	
	First()	QueryField object
	Item(*Number*)	QueryField object
	ItemByNameAndAlias(*Name*, *RecordAlias*)	QueryField object
	ItemByExpNum(*ExpNum*)	QueryField object
	Next()	QueryField object
QueryField Collection	*Property*	
	Count	Number, RO
QueryField	*Method*	
	AddTranslateExpression (*ExpressionName*)	QueryExpression object
	AddTranslateField(*FieldName*)	QueryField object
QueryField	*Properties*	
	Aggregate	Number
	ColumnNumber	Number
	Description	String, RO
	Decimal	Number, RO
	ExpNum	Number

Class Name	Methods and Properties	Returns
	Flag	Number, RO
	Format	Number
	HeadingText	String
	HeadingType	String
	HeadingUniqueFieldName	String
	Length	Number, RO
	LongName	String, RO
	Name	String, RO
	OrderByDirection	Number
	OrderByNumber	Number
	QueryRecord	QueryRecord object, RO
	RecordAlias	String, RO
	ShortName	String, RO
	TranslateEffDtLogic	String
	TranslateExpression	Expression object, RO
	TranslateField	QueryField, RO
	TranslateOption	String
	Type	String
QueryCriteria Collection	*Methods*	

Class Name	Methods and Properties	Returns
	First()	QueryCriteria object
	Item(*Number*)	QueryCriteria object
	ItemByName(*CriteriaName*)	QueryCriteria object
	Next()	QueryCriteria object
QueryCriteria Collection	*Property*	
	Count	Number, RO
QueryCriteria	*Methods*	
	AddExpr1Expression()	QueryExpression object
	AddExpr1Field(*QueryRecordAlias, FieldName*)	QueryField object
	AddExpr2Expression()	QueryExpression object
	AddExpr2Field1 (*QueryRecordAlias, FieldName*)	QueryField object
	AddExpr2Field2 (*QueryRecordAlias, FieldName*)	QueryField object
	AddExpr2List()	QueryList object
	AddExpr2Subquery()	QuerySelect object
QueryCriteria	*Properties*	
	Expr1Expression	QueryExpression object, RO
	Expr1Field	QueryField object, RO
	Expr1Type	Number

Class Name	Methods and Properties	Returns
	Expr2Constant1	String
	Expr2Constant2	String
	Expr2Expression1	QueryExpression object, RO
	Expr2Expression2	QueryExpression object, RO
	Expr2Field1	QueryField object, RO
	Expr2Field2	QueryField object, RO
	Expr2List	QueryList object, RO
	Expr2Subquery	QuerySelect object, RO
	Expr2Type	Number
	Logical	Number
	LParenLvl	Number
	Name	String, RO
	Negation	Boolean, RO
	Operator	Number
	R1ExprNum	Number
	R2ExprNum	Number
	R1ExprType	Number
	R2ExprType	Number
	RParenLvl	Number

Class Name	Methods and Properties	Returns
QueryExpression Collection	*Methods*	
	First()	QueryExpression object
	Item(*Number*)	QueryExpression object
	ItemByName(*ExpressionName*)	QueryExpression object
	Next()	QueryExpression object
QueryExpression Collection	*Property*	
	Count	Number, RO
QueryExpression	*Properties*	
	Aggregate	Number
	BindFlag	Number
	Decimal	Number
	ExpNum	Number
	Length	Number
	Name	String, RO
	OutputField	QueryField object
	RightExprFlag	Number
	SelectedField	QueryField object
	Text	String
	Type	Number

Class Name	Methods and Properties	Returns
QueryList Class	*Methods*	
	AddListValue(*Value*, *IsPrompt*)	QueryListValue object
	First()	QueryListValue object
	Item(*Number*)	QueryListValue object
	Next()	QueryListValue object
QueryList Class	*Property*	
	Count	Number, RO
QueryListValue Class	*Properties*	
	IsPrompt	Boolean, RO
	Value	String, RO
QueryRecordHierarchy Collection	*Methods*	
	First()	QueryRecordHierarchy object
	Item(*Number*)	QueryRecordHierarchy object
	ItemByName(*Name*)	QueryRecordHierarchy object
	Next()	QueryRecordHierarchy object
QueryRecordHierarchy Collection	*Property*	
	Count	Number, RO
QueryRecordHierarchy	*Properties*	
	Description	String, RO

Class Name	Methods and Properties	Returns
	Level	Number, RO
	Name	String, RO
	ParentFlag	Number, RO
Query Metadata Collection	*Methods*	
	First()	Query Metadata object
	Item(*Number*)	Query Metadata object
	ItemByName(*Name*)	Query Metadata object
	Next()	Query Metadata object
Query Metadata Collection	*Property*	
	Count	Number, RO
Query Metadata	*Properties*	
	Name	String, RO
	Value	String, RO
QueryStatistics Class	*Properties*	
	AvgExecTime	String, RO
	AvgFetchTime	String, RO
	AvgNumRows	Number, RO
	ExecCount	Number, RO
	LastExecDtTm	String, RO

Class Name	Methods and Properties	Returns
QuerySecurityProfile Class	*Properties*	
	AllowAnyJoin	Boolean, RO
	AllowDistinct	Boolean, RO
	AllowExpressions	Boolean, RO
	AllowSubqueries	Boolean, RO
	AllowUnions	Boolean, RO
	ApprovePrivateQuery	Boolean, RO
	ApprovePublicQuery	Boolean, RO
	CanCreatePublic	Boolean, RO
	CanCreateWorkFlow	Boolean, RO
	CanModifyQuery	Boolean, RO
	CanRunQuery	Boolean, RO
	CanRunToCrystal	Boolean, RO
	CanRunToExcel	Boolean, RO
	LimitUnapproved	Boolean, RO
	MaxInTreeCriteria	Boolean, RO
	MaxJoins	Boolean, RO
	MaxRowsToFetch	Boolean, RO
	MaxUnapprovedRows	Boolean, RO

Class Name	Methods and Properties	Returns
QueryDBRecord Collection	*Methods*	
	First()	QueryDBRecord object
	Item(*Number*)	QueryDBRecord object
	ItemByName(*Name*)	QueryDBRecord object
	Next()	QueryDBRecord object
QueryDBRecord Collection	*Property*	
	Count	Number, RO
QueryDBRecord Class	*Method*	
	QueryDBRecordFieldByIndex (*Index*)	QueryDBRecordField object
	QueryDBRecordFieldByName (*Name*)	QueryDBRecordField object
QueryDBRecord Class	*Properties*	
	Description	String, RO
	Name	String, RO
	QueryDBRecordFields	QueryDBRecordFields collection
	RecordHierarchy	QueryRecordHierarchy Collection, RO
QueryDBRecordField Collection	*Methods*	
	First()	QueryDBRecordField object
	Item(*Number*)	QueryDBRecordField object

Class Name	Methods and Properties	Returns
	ItemByName(*Name*)	QueryDBRecordField object
	Next()	QueryDBRecordField object
	Sort(*SortCriteria*)	
QueryDBRecordField Collection	*Property*	
	Count	Number, RO
QueryDBRecordField	*Properties*	
	Decimal	Number, RO
	Description	String, RO
	Flag	Number, RO
	Format	Number, RO
	Length	Number, RO
	LongName	String, RO
	LookupTableName	String, RO
	LookupTableRecord	QueryDBRecord object, RO
	Name	String, RO
	ShortName	String, RO
	Type	Number, RO
QueryPrompt Collection	*Methods*	
	First()	QueryPrompt object

Class Name	Methods and Properties	Returns
	Item(*Number*)	QueryPrompt object
	ItemByName(*Name*)	QueryPrompt object
	Next()	QueryPrompt object
QueryPrompt Collection	*Property*	
	Count	Number, RO
QueryPrompt Class	*Properties*	
	EditType	Number
	FieldDecimal	Number
	FieldFormat	Number
	FieldLength	Number
	FieldName	String
	FieldType	Number
	HeadingText	String
	HeadingType	Number
	LangCount	Number, RO
	Name	String, RO
	PromptRecordFieldName	String, RO
	PromptTable	String

Class Name	Methods and Properties	Returns
	UniquePromptName	String
	UseCount	Number, RO

Tree Classes Methods and Properties

Tree and tree structures are instantiated from a session object. This table contains a list of all the tree classes API plus their methods, and what a method returns, if anything.

Class	Method	Returns
Branch Collection	Item(*SetID*, *UserKeyValue*, *TreeName*, *EffDt* , *BranchName*)	Branch object
Leaf Class	Cut()	
	Delete()	
	DeleteByRange(*RangeFrom*, *RangeTo*)	
	InsertDynSib	
	InsertSib(*RangeFrom*, *RangeTo*)	Leaf object
	MoveAsChild(*Node*)	
	MoveAsChildByName(*NodeName*)	
	MoveAsSib(*Leaf*)	
	MoveAsSibByRange(*RangeFrom*, *RangeTo*)	
	PasteSib()	
Level Collection	Add(*LevelName*)	Level object
	Item*(LevelName*, *LevelNumber*)	Level object

Class	Method	Returns
	Remove()	
	Save()	
Level Class	Create(*LevelName*, *LevelNumber*)	
	Delete()	
Node Class	Branch()	
	Cut()	
	Delete()	
	DeleteByName(*NodeName*)	
	Expand(*ExpandType*)	
	InsertChildLeaf(*RangeFrom*, *RangeTo*)	Leaf object
	InsertChildNode(*NodeName*)	Node object
	InsertChildRecord(*NodeName*)	Node object
	InsertDynChildLeaf()	Leaf object
	InsertSib(*NodeName*)	Node object
	InsertSibRecord(*NodeName*)	Node object
	MoveAsChild(*Node*)	
	MoveAsChildByName(*NodeName*)	
	MoveAsSib(*Node*)	
	MoveAsSibByName(*NodeName*)	

Class	Method	Returns
	PasteChild()	
	PasteSib()	
	RefreshDescription()	Number
	Rename(*NodeName*)	
	SwitchLevel(*NewLevelNumber*)	Number
	Unbranch()	
Tree Class	Audit()	Boolean
	AuditByName(*SetID, UserKeyValue, TreeName, EffDt, BranchName*)	Boolean
	Close()	
	Copy(*FromSetId, FromUserKeyValue, FromTreeName, FromEffDt, FromBranchName, ToSetId, ToUserKeyValue, ToTreeName, ToEffDt, ToBranchName*)	Number
	Create(*SetID, UserKeyValue, TreeName, EffDt, StructureName*)	Number
	Delete(*SetID, UserKeyValue, TreeName, EffDt, BranchName*)	Number
	Exists(*SetID, UserKeyValue, TreeName, EffDt, BranchName*)	Number
	FindLeaf(RangeFrom, RangeTo)	Leaf object
	FindNextDisplayObject(*&Node, &Leaf*)	

Class	Method	Returns
	FindNode(*NodeName, Description*)	Node object
	FindPrevDisplayObject(*&Node, &Leaf*)	
	FindRoot()	Node object
	InsertRoot(*NodeName*)	
	LeafExists(*RangeFrom, RangeTo*)	Number
	Open(*SetID, UserKeyValue, TreeName, EffDt, BranchName, Update*)	Number
	OpenWholeTree(*SetID, UserKeyValue, TreeName, EffDt*)	Number
	NodeExists(*NodeName*)	Number
	Rename(*FromSetId, FromUserKeyValue, FromTreeName, FromEffDt, FromBranchName, ToTreeName*)	Number
	Save()	Number
	SaveAs(*SetID, UserKeyValue, TreeName, EffDt, BranchName*)	Number
	SaveAsDraft(*SetID, UserKeyValue, TreeName, EffDt, BranchName*)	Number
	SaveDraft()	Number
Tree Collection	Item(*SetId, UserKeyValue, TreeName, EffDt, BranchName*)	Tree object
Tree Structure	Close()	
	Copy(*FromStructId, ToStructId*)	

Class	Method	Returns
	Create(*StructId, Type*)	
	Delete(*StructId*)	
	Open(*StructId, Update*)	
	Rename(*FromStructId, ToStructId*)	
	Save()	
Tree Structure Collection	Item(*StructureID*)	

Tree and tree structures are instantiated from a session object. This table contains a list of all the tree classes plus their properties. Properties that are read-only are marked with RO (read-only). Properties that are valid for a class are marked with an "X". Also, see the footnotes following the table.

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
AllChildCount				RO		
AllChildNodeCount				RO		
AllValues					X	
AllValuesAudit			X			
AuditDetails					X	
Branches					RO	
BranchLevel					RO	
BranchName					RO	
Category					X	
CheckLeafUserData					X	

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
ChildLeafCount				RO		
ChildNodeCount				RO		
Count	X		X[2]		X[2]	X[2]
Description			X	RO	X	X
DetailComponent						X
DetailField						X
DetailMenu						X
DetailMenuBar						X
DetailMenuItem						X
DetailMultiNavigate						X
DetailPage						X
DetailRecord						X
DisplayLeaf					RO	
DisplayLevelNumber		RO		RO		
DisplayNode					RO	
DuplicateLeaves					X	
Dynamic		X				
EffDt					X	
First	X		X[2]		X[2]	X[2]

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
FirstChildLeaf				RO		
FirstChildNode				RO		
HasChildLeaves				RO		
HasChildNodes				RO		
HasChildren				RO		
HasDetailRanges					RO	
HasNextSib		RO		RO		
HasPrevSib		RO		RO		
IndirectionMethod						X
IsBranched				RO	RO	
IsChanged		RO		RO	RO	
IsCut		RO		RO		
IsDeleted		RO		RO		
IsInserted		RO		RO		
IsOpen					RO	
IsQueryTree					RO	
IsReadOnly					RO	
IsRoot				RO		
IsValid					RO	

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
IsWholeTree					RO	
KeyBranchName[1]					RO	
KeyEffDt[1]					RO	
KeyName[1]					RO	RO
KeySetId[1]					RO	
KeyUserKeyValue[1]					RO	
Last	X		X[2]		X[2]	X[2]
LastChildLeaf				RO		
LastChildNode				RO		
LeafCount					RO	
LeafOnClipboard					RO	
LevelComponent						X
LevelCount					RO	
LevelMenu						X
LevelMenuBar						X
LevelMenuItem						X
LevelNumber				X		
LevelPage						X
LevelRecord						X

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
Levels					RO	
LevelUse					X	
Name			X	X	X	X
Next	X		X[2]		X[2]	X[2]
NextSib		RO		RO		
NodeComponent						X
NodeCount					RO	
NodeField						X
NodeMenu						X
NodeMenuBar						X
NodeMenuItem						X
NodeMultiNode						X
NodeOnClipboard					RO	
NodePage						X
NodeRecord						X
NodeUserKeyField						X
Number			X			
Parent		RO		RO		
ParentLevel					RO	

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
ParentName					RO	
PerformanceMethod					X	
PerformanceSelector					X	
PerformanceSelector Option					X	
PrevSib		RO		RO		
RangeFrom		X				
RangeTo		X				
SetID					X	
SilentMode					X	
State				RO		
Status					X	
Structure					RO	
StructureName					X	
SummarySetId						X
SummaryLevel Number						X
SummaryTreeName						X
SummaryUserKey Value						X
TreeBranchName		RO	RO	RO		
TreeEffDt		RO	RO	RO		

Properties	Branch Coll	Leaf	Level	Node	Tree	Tree Structure
TreeName		RO	RO	RO		
TreeSetId		RO	RO	RO		
TreeUserKeyValue		RO	RO	RO		
TotalNodeCount					X	
Type				RO		X
UserKeyValue					X	

[1] These properties can be used with a closed tree (or tree structure.)

[2] These properties are used with the tree collection, the tree structure collection, the level collection, and the branch collection objects, not the tree, tree structure or level objects.

Mobile Classes Methods and Properties

The mobile object is accessed using the %ThisMobileObject system variable.

Class	Functions, Methods and Properties	Returns
Mobile object	*System Variable*	
	%ThisMobileObject	Mobile object
	Functions	
	GenerateMobileTree(*CIObject* [,*CIObject_property*])	
	TransferMobilePage (**MOBILEPAGE**.*PageName*, *Tab*, *CIObject*)	
Mobile object	*Methods*	

Class	Functions, Methods and Properties	Returns
	Create()	Boolean
	Find()	collection of empty mobile objects
	Get()	Boolean
	GetName()	String
	GetParent()	Mobile object
	GetPropertyAttrsByName (*propertyname*)	PropertyAttrs object
	GetPropertyByName (*propertyname*)	Depends on property. Will return a collection if *propertyname* is a collection.
	GetPropertyInfoByName (*propertyname*)	PropertyInfo object
	GetTopParent()	Mobile object
	IsNew()	Boolean
	IsNewAndNeverSaved()	Boolean
	IsSameAs(*&MobileObject*)	Boolean
	ReadBlobFromFile (*blobpropertyname*, *filename*	Boolean
	Save()	Boolean
	SetModified(*IsModified*	Boolean
	SetPropertyByName(*propertyname*)	
	ValidateEnum(*propertyname*)	Boolean

Class	Functions, Methods and Properties	Returns
	ValuesDifferFromLastSync()	Boolean
	WasSaved()	Boolean
	WriteBlobToFile (*blobpropertyname*, *filename*)	Boolean
Mobile object	*Properties*	
	oid	String, RO
	PropertyInfoCollection	PropertyInfoCollection, RO
	Version	String
PropertyInfo Collection	*Method*	
	Item(*number*)	PropertyInfo object
PropertyInfo Collection	*Property*	
	Count	Number
PropertyInfo	*Properties*	
	DefaultValue	Depends on the value, RO
	HasDefaultValue	Boolean, RO
	IsCollection	Boolean, RO
	IsRefToParent	Boolean, RO
	IsPeerReference	Boolean, RO
	IsSimpleApplicationProperty	Boolean, RO
	IsSystemProperty	Boolean, RO

Class	Functions, Methods and Properties	Returns
	Name	String, RO
	PropertyInfoCollection	PropertyInfo Collection, RO
CI Collection	*Method*	
	Item(*index*)	Mobile object
CI Collection	*Property*	
	Count	Number, RO
Data Collection	*Methods*	
	DeleteItem(*location*)	Boolean
	GetListAttrs(*propertyname*)	ListViewAttrs object
	InsertItem(*location*)	Boolean
	Item(*Index*)	Mobile object
Data Collection	*Properties*	
	Count	Number, RO
ListViewAttrs	*Method*	
	IsSet(*attribute*)	Boolean
ListViewAttrs	*Properties*	
	DetailViewLabel	String
	Label	String
	ListViewLabel	String

Class	Functions, Methods and Properties	Returns
	Visible	Boolean
PropertyAttrs	*Method*	
	IsSet(*attribute*)	Boolean
PropertyAttrs	*Properties*	
	DisplayOnly	Boolean
	InError	Boolean
	Label	String
	ShowRequiredCue	Boolean
	Visible	Boolean
PeerDefaultAttributes	*Method*	
	IsSet(*attribute*)	Boolean
PeerDefaultAttributes	*Properties*	
	DisplayOnly	Boolean
	Label	String
	Visible	Boolean

Internet Script Classes Methods and Properties

The following is summary of methods and properties for the Request, Response, and Cookies classes used with Internet Scripts (iScripts).

Class Name	Methods and Properties	Returns
Request	*Methods*	
	GetContentBody()	String
	GetCookieNames()	Array of string
	GetCookieValue(*name*)	String
	GetHeader(*name*)	String
	GetHeaderNames()	Array of string
	GetHelpURL(*HelpContext*)	String
	GetParameter(*name*)	String
	GetParameterNames()	Array of string
	GetParameterValues(*Name*)	Array of string
Request	*Properties*	
	AuthTokenDomain	String, RO
	BrowserPlatform	String, RO
	BrowserType	String, RO
	BrowserVersion	String, RO
	ByPassSignOn	Boolean, RO
	ContentURI	String, RO
	ExpireMeta	String, RO
	FullURI	String, RO

Class Name	Methods and Properties	Returns
	HTTPMethod	String, RO
	LogoutURL	String, RO
	PathInfo	String, RO
	Protocol	String, RO
	QueryString	String, RO
	RelativeURL	String, RO
	RemoteAddr	String, RO
	RemoteHost	String, RO
	RequestURI	String, RO
	Scheme	String, RO
	ServerName	String, RO
	ServerPort	String, RO
	Timeout	Integer, RO
Response	*Methods*	
	Clear()	
	CreateCookie(*name*)	Cookie object
	GetCookie(*name*)	Cookie object
	GetCookieNames()	Array of string
	GetHeader(*name*)	String

Class Name	Methods and Properties	Returns
	GetHeaderNames()	Array of string
	GetImageURL(*ImageName*)	String
	GetJavaScriptURL(HTML.*Name*)	Srting
	GetStyleSheetURL (STYLESHEET.*name*)	String
	RedirectURL(*name*)	
	SetContentType(*Type*)	
	SetHeader(*name*, *value*)	
	UseSimpleURL({*True* | *False*})	
	Write(*HTML*)	
	WriteLine(*HTML*)	
Response	*Property*	
	Charset	String
Cookie	*Properties*	
	Domain	String
	MaxAge	Number
	Name	String, RO
	Path	String
	Secure	Boolean
	Value	String

Mapping of Functions to Methods and Properties

The first column in the following table lists built-in PeopleCode functions that existed prior to PeopleTools 8.0. The second column lists the new method or property that should be used instead of the function. The existing functions still work in PeopleTools 8.0, however, they are being retained for backward compatibility only. New applications should be created using the new classes, methods, and properties.

Existing function name	Use instead
ActiveRowCount	**ActiveRowCount** Rowset property
ClearSearchDefault	**SearchDefault** Field property
ClearSearchEdit	**SearchEdit** Field property
CompareLikeFields	**CompareFields** Record method
CopyFields	**CopyFieldsTo** or **CopyChangedFieldsTo** Record method
CopyRow	**CopyTo** Row method
CurrEffDt	**DeleteEnabled** Rowset property
CurrEffRowNum	**RowNumber** Row property, in combination with the **GetCurrEffRow** Rowset method
CurrEffSeq	**EffSeq** Rowset property
CurrentRowNumber	**RowNumber** Row property
DeleteRecord	**Delete** Record method
DeleteRow	**DeleteRow** Rowset method
FetchValue	**Value** Field property
FieldChanged	**IsChanged** Field property
GetRelField	**GetRelated** Field method

Existing function name	Use instead
GetStoredFormat	**StoredFormat** Field property
Gray	**Enabled** Field property
Hide	**Visible** Field property
HideRow	**Visible** Row property
HideScroll	**HideAllRows** Rowset method
InsertRow	**InsertRow** Rowset method
IsHidden	**Visible** Row property
NextEffDt	GetNextEffRow().REC.FIELD.Value
NextRelEffDt	GetNextEffRow.()REC.FIELD.GetRelated (rec.field).value
PriorEffDt	GetPriorEffRow().REC.Field.Value
PriorRelEffDt	GetPriorEffRow().REC.FIELD.GetRelated (rec.field).value
RecordChanged	**IsChanged** Record property
RecordDeleted	**IsDeleted** Record property
RecordNew	**IsNew**Record property
RemoteCall for Application Engine	**CallAppEngine** function
RowFlush	**FlushRow** Rowset method
RowScrollSelect	**Select** Rowset method
RowScrollSelectNew	**SelectNew** Rowset method
ScrollFlush	**Flush** Rowset method

Existing function name	Use instead
ScrollSelect	**Select** Rowset method
ScrollSelectNew	**SelectNew** Rowset method
SetDefault	**SearchDefault** Field property
SetDefaultAll	**SetDefault** Rowset method
SetDefaultNext	GetNextEffRow().REC.FIELD.SetDefault()
SetDefaultNextRel	GetNextEffRow().REC.Field.GetRelated (REC.FIELD).SetDefault()
SetDefaultPrior	GetPriorEffRow().REC.FIELD.SetDefault()
SetDefaultPriorRel	GetPriorEffRow().REC.Field.GetRelated (REC.FIELD).SetDefault()
SetDisplayFormat	**DisplayFormat** Field property
SetLabel	**Label** Field property
SetSearchDefault	**SearchDefault** Field property
SetSearchEdit	**SearchEdit** Field property
SetTracePC	If using API (Session object), use Trace Setting Class properties.
SetTraceSQL	If using API (Session object), use Trace Setting Class properties.
SortScroll	**Sort** Rowset method
TotalRowCount	**RowCount** Rowset property
Ungray	**Enabled** Field property
UnHide	**Visible** Field property

Existing function name	Use instead
UnHideRow	**Visible** Row property
UnhideScroll	**ShowAllRows** Rowset method
UpdateValue	**Value** Field property

Mapping of Old Names to New Names

In PeopleTools 8.1, the names of some of the definitions in PeopleSoft Application Designer changed. The names of related built-in functions have changed accordingly.

The first table lists the old terms and new terms.

In the second table, the left column lists the old names of the functions, system variables, or reserved words. The right column lists the new names. The existing functions, system variables, and reserved words in this table still work in PeopleTools 8.0, however, they are being retained for backward compatibility only. New applications should be created using the new functions, system variables, and reserved words.

Old Term	New Term
Operator	User
Panel	Page
Panel Group	Component
Business Component	Component Interface

Deprecated Function, System Variable or Reserved Word	New Function, System Variable or Reserved Word
DoModalPanelGroup built-in function	**DoModalComponent** built-in function
IsModalPanelGroup built-in function	**IsModalComponent** built-in function
IsOperatorInClass built-in function	**IsUserInPermissionList** built-in function
PanelGroupChanged built-in function	**ComponentChanged** built-in function

Deprecated Function, System Variable or Reserved Word	New Function, System Variable or Reserved Word
ScheduleProcess built-in function	**CreateProcessRequest** built-in function
SetNextPanel built-in function	**SetNextPage** built-in function
TransferPanel built-in function	**TransferPage** built-in function
%OperatorClass System Variable	%**PrimaryPermissionList** System Variable
%OperatorID System Variable	**%UserId** System Variable
%OperatorRowLevelSecurityClass System Variable	**%RowSecurityPermissionList** System Variable
%Panel System Variable	**%Page** System Variable
%PanelGroup System Variable	**%Component** System Variable
PANEL reserved word	PAGE reserved word
PANELGROUP reserved word	COMPONENT reserved word
PanelGroup variable declaration	Component variable declaration

Business Components are now named Component Interfaces. For Component Interfaces, the old reserved word, methods, and system variables *are no longer valid*. You *must* use the new reserved word, methods, or system variables.

No Longer Valid	Use Instead
COMPONENT reserved word	COMPINTFC reserved word
GetComponent method	GetCompIntfc method
FindComponent method	FindCompIntfcs method
%Component system variable	%CompIntfc system variable

Deprecated Functions

The following functions are deprecated, and no longer work in release 8.4.

Function	Description
%MessageAgent	Used with Message Agent activities
ChDir	Used only on the client with DOS
ChDrive	Used only on the client with DOS
CheckMenuItem	Used with menu items
GetControl	Used with ActiveX controls
GetControlOccurence	Used with ActiveX controls
GetCwd	Used only on the client with DOS
GetMessageInstance	Used with messaging
GetPubContractInstance	Used with messaging
GetSelectedTreeNode	Used with dynamic trees
GetSubContractInstance	Used with messaging
GetTreeNodeParent	Used with dynamic trees
GetTreeNodeRecordName	Used with dynamic trees
RefreshTree	Used with dynamic trees
ScheduleProcess	Used with Peoplesoft Process Scheduler
UnCheckMenuItem	Used with menu items

Deprecated Methods and Properties

The following are the methods and properties that have been deprecated.

Method or Property	Use Instead
Session Class Connect method	%Session system variable

PeopleCode Syntax Cheat Sheet

The following is a description of the PeopleCode syntax, as derived from the PeopleCode parser. This description uses these "metacharacters":

[ ]	Brackets are used to indicate optional portions.
…	Ellipses are used to indicate that the preceding part (often optional) repeats an arbitrary number of times.
|	Vertical bars are used to separate alternatives. They indicate that one of the alternatives should be present. The alternatives extend only to the immediately enclosing parentheses or brackets (if any).
()	Parentheses are used to group portions of definitions to limit the scope of alternatives (|) or optionality (...).
'symbols'	Quoted characters in **keyword style** are keywords or punctuation in the language. They should appear literally, without the quotation marks.
Program	ImportList (DeclList TopStList | ClassDefn)
ImportList	[ImportDecl **';'**]...
ImportDecl	**'import'** PackageName [**':'** **'*'**]
PackageName	Id [**':'** Id]...
ClassDefn	ClassDecl [**';'** ExtDecls] [**';'** MethodDefns]
ClassDecl	**'class'** Id [Extends] [ClassPublics] [ClassPrivates] **'end-class'**
Extends	**'extends'** QualifiedId
QualifiedId	[PackageName **':'**] Id
ClassPublics	ClassPublic [**';'** ClassPublic]...
ClassPublic	MethodDecl | PropertyDecl
MethodDecl	**'method'** Id **'('** [MethodParameters] **')'** [**'returns'** PeopleCodeType]

MethodParameters	MethodParameter [**‘,’** MethodParameter]...
MethodParameter	**‘&’** Id **‘as’** PeopleCodeType [**‘out’**]
PropertyDecl	**‘property’** PeopleCodeType Id [**‘get’** [**‘set’**] | **‘readonly’**]
ClassPrivates	**‘private’** [ClassPrivate [**‘;’** ClassPrivate]...]
ClassPrivate	MethodDecl | InstanceDecl | ConstantDecl
InstanceDecl	**‘instance’** VarDeclare
ExtDecls	ExtDecl [**‘;’** ExtDecl]...
MethodDefns	MethodDefn [**‘;’** MethodDefn]...
MethodDefn	MethodMethod | GetMethod | SetMethod
MethodMethod	**‘method’** Id StList **‘end-method’**
GetMethod	**‘get’** Id StList **‘end-get’**
SetMethod	**‘set’** Id StList **‘end-set’**
TopStList	TopStmt [**‘;’** TopStmt]...
TopStmt	Stmt
StList	Stmt [**‘;’** Stmt]...
Stmt	[LValue [Assign] | If | Evaluate | While | Repeat | For | Accept | Break | Return | Exit | Error | Warning | Try | Throw | LocalDecl]
Assign	**‘=’** Expression
If	**‘if’** LogicalExpression **‘then’** StList [**‘else’** StList] **‘end-if’**
Evaluate	**‘evaluate’** Expression [WhenExpr... StList]... [**‘when-other’** StList] **‘end-evaluate’**
WhenExpr	**‘when’** [RelOp] Expression
While	**‘while’** LogicalExpression StList **‘end-while’**
Repeat	**‘repeat’** StList **‘until’** LogicalExpression
For	**‘for’** Variable **‘=’** Expression **‘to’** Expression [**‘step’** Expression] StList **‘end-for’**
Accept	**‘accept’**
Return	**‘return’** [Expression]
Break	**‘break’**
Exit	**‘exit’** [Expression]
Warning	**‘warning’** Expression
Error	**‘error’** Expression

Try	**‘try’** StList (**‘catch’** QualifiedId **‘&’** Id StList)... **‘end-try’**
Throw	**‘throw’** Expresssion
LocalDecl	**‘local’** PeopleCodeType **‘&’** Id [(**’,’ ‘&’** Id)... | **‘=’** Expression]
Logical Expression	LogicalTerm [**‘or’** LogicalTerm]...
Logical Term	Relation [**‘and’** Relation]...
Relation	**‘not’** Relation | Expression [Relop Expression]
Relop	**‘not’** Relop | **‘<’** | **‘<=’** | **‘=’** | **‘>=’** | **‘>’** | **‘!=’** | **‘<>’**
Expression	Term [(**‘+’** | **‘-’** | **‘|’**) Term]...
Term	Power [(**‘*’** | **‘/’**) Power]...
Power	Primary [**‘**’** Primary]...
Primary	LValue [**‘as’** QualifiedId] | Number | String | **‘true’** | **‘false’** | **’null’** | **‘-’** Primary | **‘create’** QualifiedId Call
LValue	LValueStart [LValueObject | LValueSubscript]...
LValueStart	Variable | Id Call | **’@’** Primary | **’%’** BuiltinVar | **‘(’** LogicalExpression **‘)’**
LValueObject	**’.’** Id [Call] | Call
LValueSubscript	**’[’** Expression [**’,’** Expression]... **’]’**
Variable	**‘&’** Id | Id [**‘.’** (Id | String)] | **‘^’**
Call	**‘(’** [Expression [**‘,’** Expression]...] **‘)’**
DecList	[Decl **‘;’**]...
Decl	**‘local’** VarDeclare | **‘function’** Function | ConstDecl | ExtDecl
VarDeclare	PeopleCodeType **‘&’** Id [**‘,’ ‘&’** Id]...
Function	Id [**‘(’** InternalParameters **‘)’**] [**‘returns’** PeopleCodeType] [**‘noexport’**] [**‘doc’** String] TopStList **‘end-function’**
InternalParameters	[InternalParameter [**‘,’** InternalParameter]...]
InternalParameter	**‘&’** Id [**‘as’** PeopleCodeType]
ConstDecl	**‘constant’ ‘&’** Id **‘=’** (Number | String | **‘true’** | **‘false’** | **’null’**)
ExtDecl	**‘global’** VarDeclare | **‘component’** VarDeclare | **‘declare’** Declare
Declare	**‘function’** Id (**‘library’** DeclareLibrary | **‘peoplecode’** DeclarePC)
DeclareLibrary	String [**‘alias’** String] [**‘(’** ExternalParameters **‘)’**] [**‘returns’** ExternalType [**‘as’** PeopleCodeType]]
ExternalParameters	[ExternalParameter [**‘,’** ExternalParameter]...]
ExternalParameter	ExternalType [**‘value’** | **‘ref’**] [**‘as’** PeopleCodeType]

DeclarePC	Variable EventType
PeopleCodeType	[**‘array’ ‘of’**]... (**‘number’** | **‘string’** | **‘date’** | **‘time’** | **‘datetime’** | **‘any’** | **‘boolean’** | **‘object’** | **‘array’** | **‘integer’** | **‘float’** | QualifiedId)
ExternalType	**‘boolean’** | **‘integer’** | **‘long’** | **‘uinteger’** | **‘ulong’** | **‘string’** | **‘lstring’** | **‘ustring’** | **‘float’** | **‘double’**

APPENDIX B

Appendix B: Viewing Trees From Application Pages

This appendix is for developers of Peoplesoft applications who want to display a tree from an application page, and enable users to select a node or leaf from the tree. It provides an overview of View Trees and discusses how to invoke View Trees from application pages.

See Also

PeopleTools PeopleBook: Peoplesoft Tree Manager

Understanding View Trees

Use a secondary 'Tree Viewer' page, (PSTREEVIEWER), to display an existing Peoplesoft tree from an application using the same HTML format as PeopleSoft Tree Manager. It enables the user to select a node or a leaf from a tree and return the selected node or leaf back to the application.

The following PeopleTools definitions are used:

- Secondary Page: PSTREEVIEWER
- Work Page: PSTREEVIEWERWRK
- Work Record: PSTREEVIEWWRK

The work record and work page are used as a way to transfer data between an application page and the secondary page. The PSTREEVIEWWRK record contains fields that define which tree to display, whether the user has selected a node or leaf, and control fields that give the application some control over the display options of the tree.

The following fields identify the specific tree to be displayed. These values should be populated by the calling application as discussed in the following table.

Field	Description
SetID	SetID of the Tree to be displayed. Required if the tree is keyed by a setID.

Field	Description
SetCntrlValue	Business unit or SetCntrlValue of the Tree to be displayed. Required if the tree is keyed by a business unit or SetCntrlValue.
Tree_Name	Name of the tree to be displayed. Required.
EffDt	Specify either the Tree's Effective Date, or the date to be used for finding the most current effective-dated version of the tree. Tree Viewer performs a maximum effective date test and displays the most current tree whose effective date is less than or equal to the EffDt value passed in the PSTREEVIEWWRK record.

Note. If a tree contains branches, they are ignored and shown as regular nodes. However, the root node displays with the branch icon to indicate that the tree does contain branches.

There are two methods of opening the PSTREEVIEWER secondary page from an application:

- Without 'MultiNode' Selection (Method A): Enables user to select, and application to receive a single node with level information, or a leaf with parent node information.
- With 'MultiNode' Selection (Method B): Enables user to select, and application to receive multiple nodes without level information. If user selects a leaf the parent node is returned.

The following fields indicate whether specific nodes or a leaf has been selected by the application. These fields can be populated by the calling application if it wants the position of a specific node or leaf, identifying it as the currently selected. The fields are also updated or populated on the Tree Viewer secondary page (PSTRREVIEWER), when the user selects a specific node or leaf and clicks the Select button.

- If the application specifies the node value and the leaf value, the search tries to find the leaf under the specified node. This is important when a tree contains duplicate leaves. If a tree does have duplicate leaves and no node is specified, the first leaf occurrence is returned.
-

Field Name	Description	Remarks
Tree_Node	The Node_Id selected. Must be an exact match to the Node_Id stored in the PSTREENODE table.	Input and/or Output
Tree_Level_Num	Level number associated with the selected node.	Output

Field Name	Description	Remarks
Tree_Level	Level name associated with the selected node.	Output
Tree_Level_Descr	Level description associated with the selected node.	Output
Leaf_Selected	Y/N flag. Indicates whether the application specified a leaf.	Input
Range_From	Range from value of the selected leaf.	Input and/or Output
Range_To	Range to value of the selected leaf.	Input and/or Output
Dynamic_Flag	Indicates whether the selected leaf is dynamic.	Output
Message_Set_Nbr	Populated in PSTREEVIEWERWRK if error occurs. For example, selected node or leaf is not found. Collapsed tree is displayed.	Output
Message_Nbr	Populated in PSTREEVIEWERWRK if error occurs. For example, specifed node or leaf is not found. Collapsed tree is displayed.	Output
Multinode	Holds list of selected nodes as a comma-separated string. Populated if Multinodeselection is set to "Y".	Output

The following fields (input) can be used to control the appearance and formatting of the Tree Viewer secondary page:

Field Name	Description
Page_Size	Determines the number of lines to be displayed on a given page. If no value is specified the default value is 60 lines per page.
Show_Leaves	Y/N flag. Controls whether the Tree Viewer displays Detail Values.
Show_Levels	Y/N flag. Controls whether the Tree Viewer should display the Level Description next to the node description.
Multinodeselection	Y/N Flag. Default = "N"

Invoking View Trees From Application Pages

This section outlines the development steps and provides some sample code to view trees from an application page. It provides two methods of how to view trees from application pages:

- Without multi-node selection.
- With multi-node selection.

To view a tree from an application page:

1. Add the PSTREEVIEWERWRK page to your component as a hidden page.
2. Add a field to a work record that will then be used as a Command Button or Hyperlink to your secondary page.

   The user selects this button or link to invoke the Tree Viewer page. You also need to add the following sample code to the FieldChange event for this field.
3. Add the Command Button or Hyperlink to the application page.
4. Add a Secondary Page control to the application page, and set the secondary page to PSTREEVIEWER.

   The Secondary Page control *must* be placed on the page at a level higher than level 3.

   In addition, the Command button or Hyperlink to invoke the PSTREEVIEWER secondary page must be placed on the same level as a secondary page control.
5. Add the application PeopleCode, which should do the following:
   a. Set the values of the Tree's key fields on the *PSTREEVIEWWRK* record.

b. Determine whether a node has been previously selected, and if so, setting the *Tree_Node* field to be the ID of the selected node:

   If a leaf has been previously selected, your code should do the following:

   — Populate the *Range_From* and *Range_To* fields with the selected leaf values

   — Set the *Tree_Node* field to the parent node

   — Set the *Leaf_Selected* field to "*Y*"

c. Set any of the display options that you want to use.

d. Display the Tree Viewer secondary page (PSTREEVIEWER) by calling the DoModal() PeopleCode function.

e. Optionally, check the return code value and storing the ID of the selected node if the user selected a node and clicked the OK or Select button.

```
/* Note: Keys of Tree are stored in &SetId,&TreeName variables.
Assume that application has Leaf selected with values stored in variables &Range⇒
From; &RangeTo and has parent node name stored in &TreeNode variable. QE_TREETEST_⇒
WRK Record holds input and received output values. */

/*Tree to open specification */
PSTREEVIEWWRK.SETID = &SETID;
PSTREEVIEWWRK.SETCNTRLVALUE = " ";
PSTREEVIEWWRK.TREE_NAME = &TREENAME;
PSTREEVIEWWRK.TREE_BRANCH = " ";
PSTREEVIEWWRK.EFFDT = %DATE; /* Get Latest Tree as of Today */

/* Tree appearance specification */
PSTREEVIEWWRK.PAGE_SIZE = 60;
PSTREEVIEWWRK.SHOW_LEAVES = "Y";
PSTREEVIEWWRK.SHOW_LEVELS = "Y";
PSTREEVIEWWRK.MULTINODESELECTION = "N";

/* Leaf input specification */
/* (Assuming QE_TREETEST_WK.LEAF_SELECTED ="Y"; */
PSTREEVIEWWRK.LEAF_SELECTED = QE_TREETEST_WRK.LEAF_SELECTED;
PSTREEVIEWWRK.TREE_NODE = &TreeNode;
PSTREEVIEWWRK.RANGE_FROM = &RangeFrom;
PSTREEVIEWWRK.RANGE_TO = &RangeTo;

/* Opening the PSTREEVIEWER secondary page */
&rslt = DoModal(Page.PSTREEVIEWER, " ", - 1, - 1);

/* populating the application Record (QE_TREETEST_WRK) with output values from⇒
 user selection in Tree */
If &rslt = 1 Then
   QE_TREETEST_WRK.TREE_NODE = PSTREEVIEWWRK.TREE_NODE;
   QE_TREETEST_WRK.TREE_LEVEL_NUM = PSTREEVIEWWRK.TREE_LEVEL_NUM;
   QE_TREETEST_WRK.TREE_LEVEL = PSTREEVIEWWRK.TREE_LEVEL;
   QE_TREETEST_WRK.TREE_LEVEL_DESCR = PSTREEVIEWWRK.TREE_LEVEL_DESCR;
```

```
   QE_TREETEST_WRK.RANGE_FROM = PSTREEVIEWWRK.RANGE_FROM;
   QE_TREETEST_WRK.RANGE_TO = PSTREEVIEWWRK.RANGE_TO;
   QE_TREETEST_WRK.DYNAMIC_FLAG = PSTREEVIEWWRK.DYNAMIC_FLAG;
   QE_TREETEST_WRK.MESSAGE_SET_NBR = PSTREEVIEWWRK.MESSAGE_SET_NBR;
   QE_TREETEST_WRK.MESSAGE_NBR = PSTREEVIEWWRK.MESSAGE_NBR;
End-If;
```

The following is the sample PeopleCode (Method A), which would be part of the FieldChange event triggered from a Command Button or Hyperlink command on the application page:

In some cases, you may need to use the Component Level Record variable *&cPSTREEVIEWWRK* to set values for the tree. For example, if the application added the Tree Viewer secondary page to the application's secondary page and cannot reach the record from the component buffer. The following is the sample PeopleCode illustrating the use of the variable:

```
Component Record &cPSTREEVIEWWRK;
Component boolean &gbShowTreeLeaves;
Local number &rslt;

/* opening the Tree Viewer secondary page */

&cPSTREEVIEWWRK = CreateRecord(Record.PSTREEVIEWWRK);

&cPSTREEVIEWWRK.SETID.Value = &SETID;
&cPSTREEVIEWWRK.SETCNTRLVALUE.Value = " ";
&cPSTREEVIEWWRK.TREE_NAME.Value = &TREENAME;
&cPSTREEVIEWWRK.TREE_BRANCH.Value = " ";
&cPSTREEVIEWWRK.EFFDT.Value = %DATE; /* Get Latest Tree as of Today */;
&cPSTREEVIEWWRK.PAGE_SIZE.Value = 60;
&cPSTREEVIEWWRK.SHOW_LEVELS.Value = "Y";
&cPSTREEVIEWWRK.MULTINODESELECTION.Value = "N";

If &gbShowTreeLeaves Then
   &cPSTREEVIEWWRK.SHOW_LEAVES.Value = "Y";
Else
   &cPSTREEVIEWWRK.SHOW_LEAVES.Value = "N";
End-If;

&rslt = DoModal(Page.PSTREEVIEWER, "", - 1, - 1);

/* reading output value in a case when Component Level Record variable &c⇒
PSTREEVIEWERWRK is used. */

If &rslt = 1 Then
   QE_TREETEST_WRK.TREE_NODE = &cPSTREEVIEWWRK.TREE_NODE.value;
   QE_TREETEST_WRK.TREE_LEVEL_NUM = &cPSTREEVIEWWRK.TREE_LEVEL_NUM.value;
   QE_TREETEST_WRK.TREE_LEVEL = &cPSTREEVIEWWRK.TREE_LEVEL.value;
   QE_TREETEST_WRK.TREE_LEVEL_DESCR = &cPSTREEVIEWWRK.TREE_LEVEL_DESCR.value;
   QE_TREETEST_WRK.RANGE_FROM = &cPSTREEVIEWWRK.RANGE_FROM.value;
   QE_TREETEST_WRK.RANGE_TO = &cPSTREEVIEWWRK.RANGE_TO.value;
```

```
      QE_TREETEST_WRK.DYNAMIC_FLAG = &cPSTREEVIEWWRK.DYNAMIC_FLAG.value;
      QE_TREETEST_WRK.MESSAGE_SET_NBR = &cPSTREEVIEWWRK.MESSAGE_SET_NBR.value;
      QE_TREETEST_WRK.MESSAGE_NBR = &cPSTREEVIEWWRK.MESSAGE_NBR.value;

      EndModal(1);

   Else
      EndModal(0);

   End-If;
```

Note. The name of the variable *&cPSTREEVIEWWRK* is hard-coded and should not be changed.

The segment of code in *italics* reads the results that came from the tree. (Node or leaf selected).

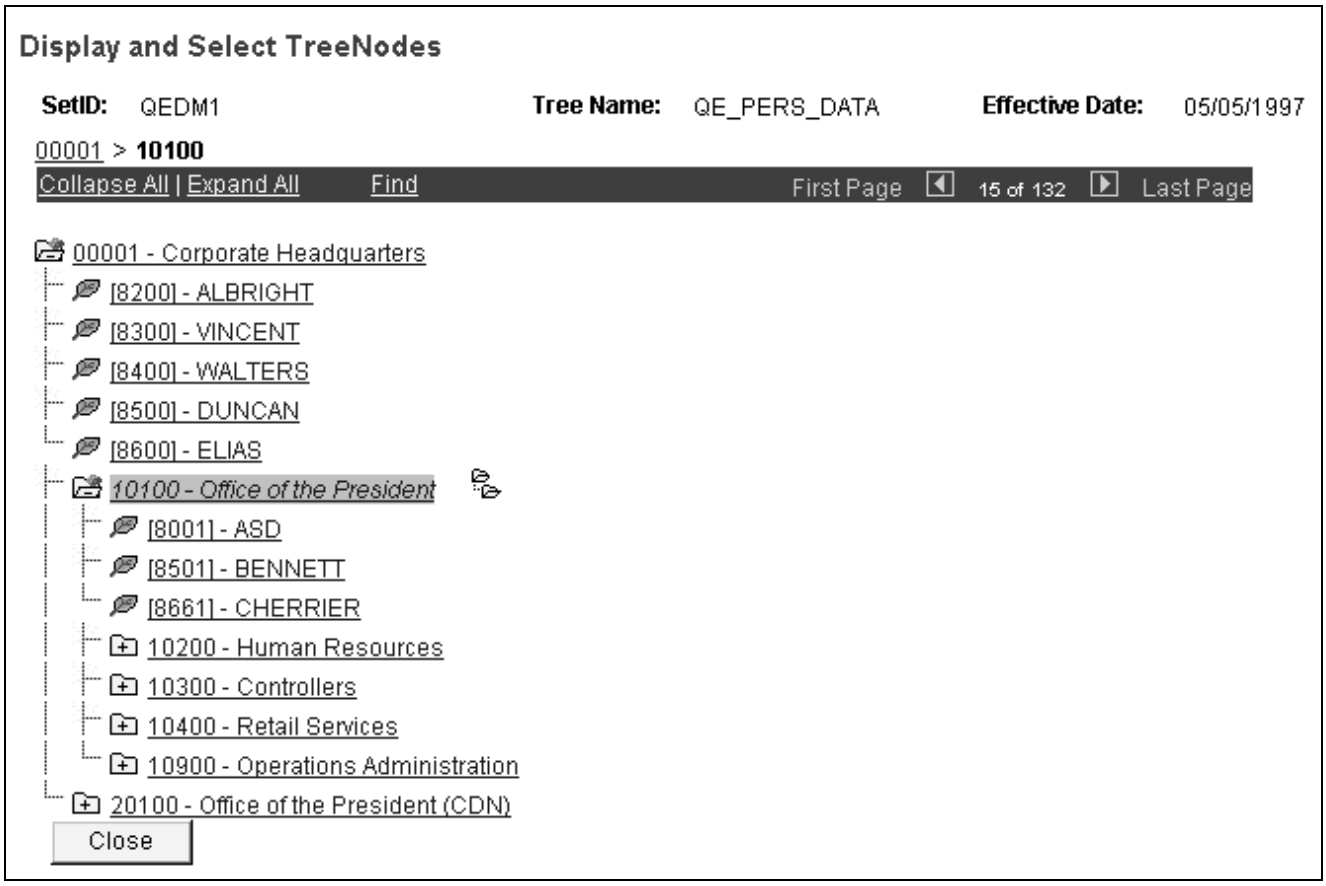
View Trees From Application Example−Without Multi-Node Selection (Method A)

An example of an application that uses the Tree Viewer secondary page (PSTREEVIEWER), with the Multinodeselection flag set to "**N**", is the Using Copy/Delete Tree (PSTREEMAINT) component. Navigate to the component, select any tree and click the View button.

See .

Understanding the Tree Viewer page

Page Name	Object Name	Navigation	Usage
Copy/Delete Tree - Tree Viewer	PSTREEVIEWER	Tree Manager, Tree Utilities, Copy/Delete Tree	This page is using the Tree Viewer secondary page



Copy/Delete Tree - Tree Viewer page

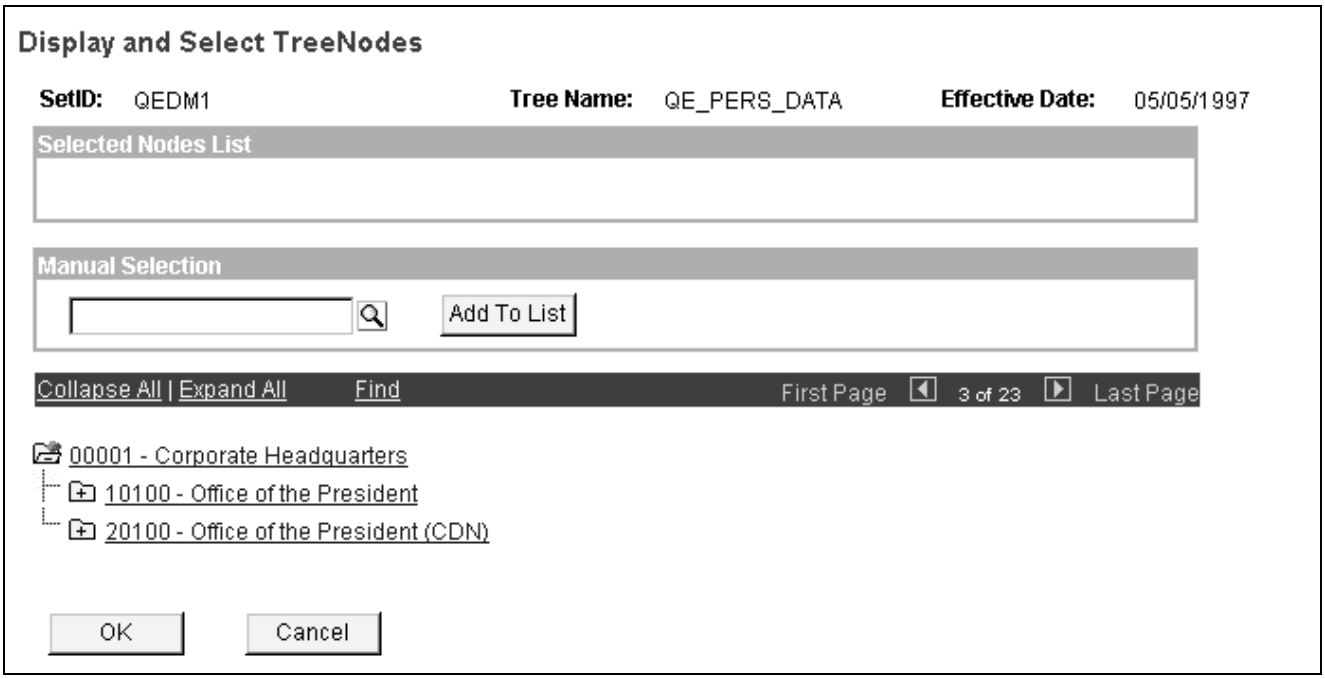
View Trees From Application Example−With Multi-Node Selection (Method B)

An example of an application that uses the PSTREEVIEWER secondary page, with the Multinodeselection flag set to "Y", is the Using Query Manager component. Navigate to the Query Manager component, select any query and click the Add Criteria Image. The Edit Criteria Properties page opens. Select **field** as the first expression then select **in tree** as the condition type. Click the **New Node List** link to display the Select Tree page. Click the name of the desired tree.

See .

Understanding the Display and Select TreeNodes Page

Page Name	Object Name	Navigation	Usage
Select TreeNodes	PSTREEVIEWER	Reporting Tools, Query, Query Manager	Use this page to create a list of values for Query to compare to the value from the first expression.



Select TreeNodes page

The Tree Viewer secondary page (PSTREEVIEWER) in Method B has a frame that holds the Selected Nodes List with action buttons associated with each selected node.

This page is used to select the set of nodes, get them back to the calling application, (Query Manager), and use the list of nodes as Query criteria.

To select tree nodes:

1. Highlight the desired tree node and click the Add Node icon.
2. If you know the name of the desired node you can use the manual selection option. Enter the name of the desired node or use the Lookup button to select from a list of available nodes. Click on the Add to List button to add the node to the list.
3. The nodes you entered or selected from the list appear in the *Selected Nodes* drop-down list box.
4. Delete nodes from the list by clicking the Remove Node From List icon.
5. Display the selected node in the Tree by clicking the Find icon.
6. Click OK when the list is complete to close the *Display and Select TreeNodes* (PSTREEVIEWER secondary page).

   The selected tree setID, tree name, effective date, and selected nodes display in the Select Tree Node List dialog box.

   The list of selected nodes can be read from the Multinode field of the PSTREEVIEWWRK work record as a comma-separated string. The string can be parsed to get the node names.

Glossary of PeopleSoft Terms

absence entitlement This element defines rules for granting paid time off for valid absences, such as sick time, vacation, and maternity leave. An absence entitlement element defines the entitlement amount, frequency, and entitlement period.

absence take This element defines the conditions that must be met before a payee is entitled to take paid time off.

account You use an account code to record and summarize financial transactions as expenditures, revenues, assets, or liabilities balances. The use of this delivered PeopleSoft ChartField is typically defined when you implement PeopleSoft General Ledger.

accounting class In PeopleSoft Enterprise Performance Management, the accounting class defines how a resource is treated for generally accepted accounting practices. The Inventory class indicates whether a resource becomes part of a balance sheet account, such as inventory or fixed assets, while the Non-inventory class indicates that the resource is treated as an expense of the period during which it occurs.

accounting date The accounting date indicates when a transaction is recognized, as opposed to the date the transaction actually occurred. The accounting date and transaction date can be the same. The accounting date determines the period in the general ledger to which the transaction is to be posted. You can only select an accounting date that falls within an open period in the ledger to which you are posting. The accounting date for an item is normally the invoice date.

accounting entry A set of related debits and credits. An accounting entry is made up of multiple accounting lines. In most PeopleSoft applications, accounting entries are always balanced (debits equal credits). Accounting entries are created to record accruals, payments, payment cancellations, manual closures, project activities in the general ledger, and so forth, depending on the application.

accounting split The accounting split method indicates how expenses are allocated or divided among one or more sets of accounting ChartFields.

accumulator You use an accumulator to store cumulative values of defined items as they are processed. You can accumulate a single value over time or multiple values over time. For example, an accumulator could consist of all voluntary deductions, or all company deductions, enabling you to accumulate amounts. It allows total flexibility for time periods and values accumulated.

action reason The reason an employee's job or employment information is updated. The action reason is entered in two parts: a personnel action, such as a promotion, termination, or change from one pay group to anotherand a reason for that action. Action reasons are used by PeopleSoft Human Resources, PeopleSoft Benefits Administration, PeopleSoft Stock Administration, and the COBRA Administration feature of the Base Benefits business process.

activity In PeopleSoft Enterprise Learning Management, an instance of a catalog item delivery methodit may also be called a class. The activity defines such things as meeting times and locations, instructors, reserved equipment and materials, and detailed costs that are associated with the offering, enrollment limits and deadlines, and waitlisting capacities.

allocation rule In PeopleSoft Enterprise Incentive Management, an expression within compensation plans that enables the system to assign transactions to nodes and participants. During transaction allocation, the allocation engine traverses the compensation structure

	from the current node to the root node, checking each node for plans that contain allocation rules.
alternate account	A feature in PeopleSoft General Ledger that enables you to create a statutory chart of accounts and enter statutory account transactions at the detail transaction level, as required for recording and reporting by some national governments.
application agent	An application agent is an online agent that is loaded into memory with a PeopleSoft page. It detects when a business rule has been triggered and determines the appropriate action.
asset class	An asset group used for reporting purposes. It can be used in conjunction with the asset category to refine asset classification.
attachment	In PeopleSoft Enterprise Learning Management, nonsystem-defined electronic material that supplements a learning resource, such as an equipment items user handbook or the site map of a large facility.
background process	In PeopleSoft, background processes are executed through process-specific COBOL programs and run outside the Windows environment.
benchmark job	In PeopleSoft Workforce Analytics, a benchmark job is a job code for which there is corresponding salary survey data from published, third-party sources.
branch	A tree node that rolls up to nodes above it in the hierarchy, as defined in PeopleSoft Tree Manager.
budgetary account only	An account used by the system only and not by users; this type of account does not accept transactions. You can only budget with this account. Formerly called system-maintained account.
budget check	In commitment control, the processing of source transactions against control budget ledgers, to see if they pass, fail, or pass with a warning.
budget control	In commitment control, budget control ensures that commitments and expenditures don't exceed budgets. It enables you to track transactions against corresponding budgets and terminate a document's cycle if the defined budget conditions are not met. For example, you can prevent a purchase order from being dispatched to a vendor if there are insufficient funds in the related budget to support it.
budget period	The interval of time (such as 12 months or 4 quarters) into which a period is divided for budgetary and reporting purposes. The ChartField allows maximum flexibility to define operational accounting time periods without restriction to only one calendar.
business event	In PeopleSoft Sales Incentive Management, an original business transaction or activity that may justify the creation of a PeopleSoft Enterprise Incentive Management event (a sale, for example).
catalog item	In PeopleSoft Enterprise Learning Management, a specific topic that a learner can study and have tracked. For example, Introduction to Microsoft Word. A catalog item contains general information about the topic and includes a course code, description, categorization, keywords, and delivery methods.
category	In PeopleSoft Enterprise Learning Management, a way to classify catalog items so that users can easily browse and search relevant entries in the learning catalog. Categories can be hierarchical.
ChartField	A field that stores a chart of accounts, resources, and so on, depending on the PeopleSoft application. ChartField values represent individual account numbers, department codes, and so forth.
ChartField balancing	You can require specific ChartFields to match up (balance) on the debit and the credit side of a transaction.

ChartField combination edit	The process of editing journal lines for valid ChartField combinations based on user-defined rules.
ChartKey	One or more fields that uniquely identify each row in a table. Some tables contain only one field as the key, while others require a combination.
child	In PeopleSoft Tree Manager trees, a child is a node or detail on a tree linked to another, higher-level node (referred to as the parent). Child nodes can be rolled up into the parent. A node can be a child and a parent at the same time depending on its location within the tree.
Class ChartField	A ChartField value that identifies a unique appropriation budget key when you combine it with a fund, department ID, and program code, as well as a budget period. Formerly called *sub-classification.*
clone	In PeopleCode, to make a unique copy. In contrast, to *copy* may mean making a new reference to an object, so if the underlying object is changed, both the copy and the original change.
collection	To make a set of documents available for searching in Verity, you must first create at least one collection. A collection is set of directories and files that allow search application users to use the Verity search engine to quickly find and display source documents that match search criteria. A collection is a set of statistics and pointers to the source documents, stored in a proprietary format on a file server. Because a collection can only store information for a single location, PeopleSoft maintains a set of collections (one per language code) for each search index object.
compensation object	In PeopleSoft Enterprise Incentive Management, a node within a compensation structure. Compensation objects are the building blocks that make up a compensation structure's hierarchical representation.
compensation structure	In PeopleSoft Enterprise Incentive Management, a hierarchical relationship of compensation objects that represents the compensation-related relationship between the objects.
configuration parameter catalog	Used to configure an external system with PeopleSoft. For example, a configuration parameter catalog might set up configuration and communication parameters for an external server.
configuration plan	In PeopleSoft Enterprise Incentive Management, configuration plans hold allocation information for common variables (not incentive rules) and are attached to a node without a participant. Configuration plans are not processed by transactions.
content reference	Content references are pointers to content registered in the portal registry. These are typically either URLs or iScripts. Content references fall into three categories: target content, templates, and template pagelets.
context	In PeopleSoft Enterprise Incentive Management, a mechanism that is used to determine the scope of a processing run. PeopleSoft Enterprise Incentive Management uses three types of context: plan, period, and run-level.
corporate account	Equivalent to the Account ChartField. Distinguishes between the chart of accounts typically used to record and report financial information for management, stockholders, and the general public, as opposed to a chart of statutory (alternate) accounts required by a regulatory authority for recording and reporting financial information.
cost profile	A combination of a receipt cost method, a cost flow, and a deplete cost method. A profile is associated with a cost book and determines how items in that book are valued, as well as how the material movement of the item is valued for the book.
cost row	A cost transaction and amount for a set of ChartFields.

data acquisition In PeopleSoft Enterprise Incentive Management, the process during which raw business transactions are acquired from external source systems and fed into the operational data store (ODS).

data elements Data elements, at their simplest level, define a subset of data and the rules by which to group them.

For Workforce Analytics, data elements are rules that tell the system what measures to retrieve about your workforce groups.

data row Contains the entries for each field in a table. To identify each data row uniquely, PeopleSoft applications use a key consisting of one or more fields in the table.

data validation In PeopleSoft Enterprise Incentive Management, a process of validating and cleansing the feed data to resolve conflicts and make the data processable.

DAT file This text file, used with the Verity search engine, contains all of the information from documents that are searchable but not returned in the results list.

delivery method In PeopleSoft Enterprise Learning Management, identifies a learning activitys delivery method type. An activity can have one or more delivery methods.

delivery method type In PeopleSoft Enterprise Learning Management, specifies a method that your organization uses to deliver learning activities, for example, scheduled or self-paced learning.

distribution The process of assigning values to ChartFields. A distribution is a string of ChartField values assigned to items, payments, and budget amounts.

double byte character If youre working with Japanese or other Asian employees, you can enter the employee's name using double-byte characters. The standard double byte character set name format in PeopleSoft applications is: [last name] space [first name].

dynamic tree A tree that takes its detail valuesdynamic detailsdirectly from a table in the database, rather than from a range of values entered by the user.

edit table A table in the database that has its own record definition, such as the Department table. As fields are entered into a PeopleSoft application, they can be validated against an edit table to ensure data integrity throughout the system.

effective date A method of dating information in PeopleSoft applications. You can predate information to add historical data to your system, or postdate information in order to enter it before it actually goes into effect. By using effective dates, you don't delete values; you enter a new value with a current effective date.

EIM job Abbreviation for *Enterprise Incentive Management job.* In PeopleSoft Enterprise Incentive Management, a collection of job steps that corresponds to the steps in an organizations compensation-related business process. An EIM job can be stopped to allow manual changes or corrections to be applied between steps, and then resumed from where it left off, continuing with the next step. A run can also be restarted or rolled back.

EIM ledger Abbreviation for *Enterprise Incentive Management ledger.* In PeopleSoft Enterprise Incentive Management, an object to handle incremental result gathering within the scope of a participant. The ledger captures a result set with all of the appropriate traces to the data origin and to the processing steps of which it is a result.

equipment In PeopleSoft Enterprise Learning Management, resource items that can be assigned to a training facility, to a specific training room, or directly to an activity session. Equipment items are generally items that are used (sometimes for a fee) and returned after the activity is complete.

event Events are predefined points either in the application processor flow or in the program flow. As each point is encountered, the event activates each component, triggering any PeopleCode program associated with that component and that event. Examples of events are FieldChange, SavePreChange, and OnRouteSubscription. In PeopleSoft Human Resources, *event* also refers to incidents that affect benefits eligibility.

event propagation process In PeopleSoft Sales Incentive Management, a process that determines, through logic, the propagation of an original PeopleSoft Enterprise Incentive Management event and creates a derivative (duplicate) of the original event to be processed by other objects. Sales Incentive Management uses this mechanism to implement splits, roll-ups, and so on. Event propagation determines who receives the credit.

external system In PeopleSoft, any system that is not directly compiled with PeopleTools servers.

fact In PeopleSoft applications, facts are numeric data values from fields from a source database as well as an analytic application. A fact can be anything you want to measure your business by, for example, revenue, actual, budget data, or sales numbers. A fact is stored on a fact table.

filter In PeopleSoft applications, a filter creates a subset of information. Filters are used in templates to limit your information from a pick list of attribute values.

generic process type In PeopleSoft Process Scheduler, process types are identified by a generic process type. For example, the generic process type SQR includes all SQR process types, such as SQR process and SQR report.

group Any set of records associated under a single name or variable in order to run calculations in PeopleSoft business processes. In PeopleSoft Time and Labor, for example, employees are placed in groups for time reporting purposes.

homepage Users can personalize the homepage, or the page that first appears when they access the portal.

incentive object In PeopleSoft Enterprise Incentive Management, the incentive-related objects that define and support the PeopleSoft Enterprise Incentive Management calculation process and results, such as plan templates, plans, results data, user interaction objects, and so on.

incentive rule In PeopleSoft Sales Incentive Management, the commands that act on transactions and turn them into compensation. A rule is one part in the process of turning a transaction into compensation.

key One or more fields that uniquely identify each row in a table. Some tables contain only one field as the key, while others require a combination.

learner group In PeopleSoft Enterprise Learning Management, a group of learners within the same learning environment that share the same attributes, such as department or job code.

learning activity See *activity.*

learning history In PeopleSoft Enterprise Learning Management, a self-service repository for all of a learner's completed learning activities.

learning plan In PeopleSoft Enterprise Learning Management, a self-service repository for all of a learner's planned and in-progress learning activities.

ledger mapping You use ledger mapping to relate expense data from general ledger accounts to resource objects. Multiple ledger line items can be mapped to one or more resource IDs. You can also use ledger mapping to map dollar amounts (referred to as *rates*) to business units. You can map the amounts in two different ways: an actual amount that represents actual costs of the accounting period, or a budgeted amount that can be used to calculate the capacity rates as well as budgeted model results. In PeopleSoft Enterprise Warehouse, you can map general ledger accounts to the EW Ledger table.

level	A section of a tree that organizes groups of nodes.
library section	In PeopleSoft Enterprise Incentive Management, a section that is defined in a plan (or template) and that is available for other plans to share. Changes to a library section are reflected in all plans that use it.
linked section	In PeopleSoft Enterprise Incentive Management, a section that is defined in a plan template but appears in a plan. Changes to linked sections propagate to plans using that section.
linked variable	In PeopleSoft Enterprise Incentive Management, a variable that is defined and maintained in a plan template and that also appears in a plan. Changes to linked variables propagate to plans using that variable.
load	The feature that initiates a process to automatically load information into a PeopleSoft applicationfor example, populating the PeopleSoft Benefits database with plan-level election information.
local functionality	In PeopleSoft HRMS, the set of information that is available for a specific country. You can access this information when you click the appropriate country flag in the global window, or when you access it by a local country menu.
location	Locations enable you to indicate the different types of addressesfor a company, for example, one address to receive bills, another for shipping, a third for postal deliveries, and a separate street address. Each address has a different location number. The primary locationindicated by a *1*is the address you use most often and may be different from the main address.
market template	In PeopleSoft Enterprise Incentive Management, additional functionality that is specific to a given market or industry and is built on top of a product category.
material	In PeopleSoft Enterprise Learning Management, a resource item that can be assigned to the sessions of an activity. Material items are generally consumed during the duration of an activity and not returned, and they may have an associated cost.
message definition	An object definition specified in PeopleSoft Application Designer that contains message information for PeopleSoft Application Messaging.
meta-SQL	Meta-SQL constructs expand into platform-specific SQL substrings. They are used in functions that pass SQL strings, such as in SQL objects, the SQLExec function, and PeopleSoft Application Engine programs.
metastring	Metastrings are special expressions included in SQL string literals. The metastrings, prefixed with a percent (%) symbol, are included directly in the string literals. They expand at run time into an appropriate substring for the current database platform.
multibook	Processes in PeopleSoft applications that can create both application entries and general ledgers denominated in more than one currency.
multicurrency	The ability to process transactions in a currency other than the business unit's base currency.
objective	In PeopleSoft Enterprise Learning Management, an individual's learning goal. An example of a learning goal is a competency gap.
override	In PeopleSoft Enterprise Incentive Management, the ability to make a change to a plan that applies to only one plan context.
pagelet	Each block of content on the homepage is called a pagelet. These pagelets display summary information within a small rectangular area on the page. The pagelet provide users with a snapshot of their most relevant PeopleSoft and non-PeopleSoft content.

parent node A tree node linked to lower-level nodes or details that roll up into it. A node can be a parent and a child at the same time, depending on its location within the tree.

participant In PeopleSoft Enterprise Incentive Management, participants are recipients of the incentive compensation calculation process.

participant object Each participant object may be related to one or more compensation objects.

See also *participant object.*

payout In PeopleSoft Enterprise Incentive Management, the resulting incentive plan computation that is provided to payroll.

PeopleCode PeopleCode is a proprietary language, executed by the PeopleSoft application processor. PeopleCode generates results based upon existing data or user actions. By using business interlink objects, external services are available to all PeopleSoft applications wherever PeopleCode can be executed.

PeopleCode event An action that a user takes upon an object, usually a record field, that is referenced within a PeopleSoft page.

PeopleSoft Internet Architecture The fundamental architecture on which PeopleSoft 8 applications are constructed, consisting of an RDBMS, an application server, a Web server, and a browser.

performance measurement In PeopleSoft Enterprise Incentive Management, a variable used to store data (similar to an aggregator, but without a predefined formula) within the scope of an incentive plan. Performance measures are associated with a plan calendar, territory, and participant. Performance measurements are used for quota calculation and reporting.

period context In PeopleSoft Enterprise Incentive Management, because a participant typically uses the same compensation plan for multiple periods, the period context associates a plan context with a specific calendar period and fiscal year. The period context references the associated plan context, thus forming a chain. Each plan context has a corresponding set of period contexts.

per seat cost In PeopleSoft Enterprise Learning Management, the cost per learner, based on the total activity costs divided by either minimum attendees or maximum attendees. Organizations use this cost to price PeopleSoft Enterprise Learning Management activities.

plan In PeopleSoft Sales Incentive Management, a collection of allocation rules, variables, steps, sections, and incentive rules that instruct the PeopleSoft Enterprise Incentive Management engine in how to process transactions.

plan context In PeopleSoft Enterprise Incentive Management, correlates a participant with the compensation plan and node to which the participant is assigned, enabling the PeopleSoft Enterprise Incentive Management system to find anything that is associated with the node and that is required to perform compensation processing. Each participant, node, and plan combination represents a unique plan contextif three participants are on a compensation structure, each has a different plan context. Configuration plans are identified by plan contexts and are associated with the participants that refer to them.

plan section In PeopleSoft Enterprise Incentive Management, a segment of a plan that handles a specific type of event processing.

plan template In PeopleSoft Enterprise Incentive Management, the base from which a plan is created. A plan template contains common sections and variables that are inherited by all plans that are created from the template. A template may contain steps and sections that are not visible in the plan definition.

portal registry In PeopleSoft applications, the portal registry is a tree-like structure in which content references are organized, classified, and registered. It is a central repository that

	defines both the structure and content of a portal through a hierarchical, tree-like structure of folders useful for organizing and securing content references.
private view	A user-defined view that is available only to the user who created it.
process	See *Batch Processes*.
process definition	Process definitions define each run request.
process instance	A unique number that identifies each process request. This value is automatically incremented and assigned to each requested process when the process is submitted to run.
process job	You can link process definitions into a job request and process each request serially or in parallel. You can also initiate subsequent processes based on the return code from each prior request.
process request	A single run request, such as an SQR, a COBOL program, or a Crystal report that you run through PeopleSoft Process Scheduler.
process run control	A PeopleTools variable used to retain PeopleSoft Process Scheduler values needed at runtime for all requests that reference a run control ID. Do not confuse these with application run controls, which may be defined with the same run control ID, but only contain information specific to a given application process request.
product category	In PeopleSoft Enterprise Incentive Management, indicates an application in the Enterprise Incentive Management suite of products. Each transaction in the PeopleSoft Enterprise Incentive Management system is associated with a product category.
publishing	In PeopleSoft Enterprise Incentive Management, a stage in processing that makes incentive-related results available to participants.
record definition	A logical grouping of data elements.
record field	A field within a record definition.
record group	A set of logically and functionally related control tables and views. Record groups help enable TableSet sharing, which eliminates redundant data entry. Record groups ensure that TableSet sharing is applied consistently across all related tables and views.
record input VAT flag	Abbreviation for *record input value-added tax flag.* Within PeopleSoft Purchasing, Payables, and General Ledger, this flag indicates that you are recording input VAT on the transaction. This flag, in conjunction with the record output VAT flag, is used to determine the accounting entries created for a transaction and to determine how a transaction is reported on the VAT return. For all cases within Purchasing and Payables where VAT information is tracked on a transaction, this flag is set to Yes. This flag is not used in PeopleSoft Order Management, Billing, or Receivables, where it is assumed that you are always recording only output VAT, or in PeopleSoft Expenses, where it is assumed that you are always recording only input VAT.
record output VAT flag	Abbreviation for *record output value-added tax flag.* See *record input VAT flag.*
reference data	In PeopleSoft Sales Incentive Management, system objects that represent the sales organization, such as territories, participants, products, customers, channels, and so on.
reference object	In PeopleSoft Enterprise Incentive Management, this dimension-type object further defines the business. Reference objects can have their own hierarchy (for example, product tree, customer tree, industry tree, and geography tree).
reference transaction	In commitment control, a reference transaction is a source transaction that is referenced by a higher-level (and usually later) source transaction, in order to automatically reverse all or part of the referenced transaction's budget-checked

amount. This avoids duplicate postings during the sequential entry of the transaction at different commitment levels. For example, the amount of an encumbrance transaction (such as a purchase order) will, when checked and recorded against a budget, cause the system to concurrently reference and relieve all or part of the amount of a corresponding pre-encumbrance transaction, such as a purchase requisition.

relationship object In PeopleSoft Enterprise Incentive Management, these objects further define a compensation structure to resolve transactions by establishing associations between compensation objects and business objects.

results management process In PeopleSoft Sales Incentive Management, the process during which compensation administrators may review processing results, manually change transactions, process draws, update and review payouts, process approvals, and accumulate and push payments to the EIM ledger.

role user A PeopleSoft Workflow user. A person's role user ID serves much the same purpose as a user ID does in other parts of the system. PeopleSoft Workflow uses role user IDs to determine how to route worklist items to users (through an email address, for example) and to track the roles that users play in the workflow. Role users do not need PeopleSoft user IDs.

role Describes how people fit into PeopleSoft Workflow. A role is a class of users who perform the same type of work, such as clerks or managers. Your business rules typically specify what user role needs to do an activity.

roll up In a tree, to roll up is to total sums based on the information hierarchy.

routing Connects activities in PeopleSoft Workflow. Routings specify where the information goes and what form it takesemail message, electronic form, or worklist entry.

run control A run control is a type of online page that is used to begin a process, such as the batch processing of a payroll run. Run control pages generally start a program that manipulates data.

run control ID A unique ID to associate each user with his or her own run control table entries.

run-level context In PeopleSoft Enterprise Incentive Management, associates a particular run (and batch ID) with a period context and plan context. Every plan context that participates in a run has a separate run-level context. Because a run cannot span periods, only one run-level context is associated with each plan context.

search query You use this set of objects to pass a query string and operators to the search engine. The search index returns a set of matching results with keys to the source documents.

section In PeopleSoft Enterprise Incentive Management, a collection of incentive rules that operate on transactions of a specific type. Sections enable plans to be segmented to process logical events in different sections.

security event In commitment control, security events trigger security authorization checking, such as budget entries, transfers, and adjustments; exception overrides and notifications; and inquiries.

self-service application Self-service refers to PeopleSoft applications that are accessed by end users with a browser.

session In PeopleSoft Enterprise Learning Management, a single meeting day of an activity (that is, the period of time between start and finish times within a day). The session stores the specific date, location, meeting time, and instructor. Sessions are used for scheduled training.

session template In PeopleSoft Enterprise Learning Management, enables you to set up common activity characteristics that may be reused while scheduling a PeopleSoft Enterprise Learning Management activitycharacteristics such as days of the week, start and end

	times, facility and room assignments, instructors, and equipment. A session pattern template can be attached to an activity that is being scheduled. Attaching a template to an activity causes all of the default template information to populate the activity session pattern.
setup relationship	In PeopleSoft Enterprise Incentive Management, a relationship object type that associates a configuration plan with any structure node.
sibling	A tree node at the same level as another node, where both roll up into the same parent. A node can be a sibling, parent, and child all at the same time, depending on its location in the tree.
single signon	With single signon, users can, after being authenticated by a PeopleSoft application server, access a second PeopleSoft application server without entering a user ID or password.
source transaction	In commitment control, any transaction generated in a PeopleSoft or third-party application that is integrated with commitment control and which can be checked against commitment control budgets. For example, a pre-encumbrance, encumbrance, expenditure, recognized revenue, or collected revenue transaction.
SpeedChart	A user-defined shorthand key that designates several ChartKeys to be used for voucher entry. Percentages can optionally be related to each ChartKey in a SpeedChart definition.
SpeedType	A code representing a combination of ChartField values. SpeedTypes simplify the entry of ChartFields commonly used together.
SQR	See *Structured Query Report (SQR).*
statutory account	Account required by a regulatory authority for recording and reporting financial results. In PeopleSoft, this is equivalent to the Alternate Account (ALTACCT) ChartField.
step	In PeopleSoft Sales Incentive Management, a collection of sections in a plan. Each step corresponds to a step in the job run.
Structured Query Report (SQR)	A type of printed or displayed report generated from data extracted from a PeopleSoft SQL-based relational database. PeopleSoft applications provide a variety of standard SQRs that summarize table information and data. You can use these reports as is, customize them, or create your own.
Summary ChartField	You use summary ChartFields to create summary ledgers that roll up detail amounts based on specific detail values or on selected tree nodes. When detail values are summarized using tree nodes, summary ChartFields must be used in the summary ledger data record to accommodate the maximum length of a node name (20 characters).
summary ledger	An accounting feature used primarily in allocations, inquiries, and PS/nVision reporting to store combined account balances from detail ledgers. Summary ledgers increase speed and efficiency of reporting by eliminating the need to summarize detail ledger balances each time a report is requested. Instead, detail balances are summarized in a background process according to user-specified criteria and stored on summary ledgers. The summary ledgers are then accessed directly for reporting.
summary tree	A tree used to roll up accounts for each type of report in summary ledgers. Summary trees enable you to define trees on trees. In a summary tree, the detail values are really nodes on a detail tree or another summary tree (known as the *basis* tree). A summary tree structure specifies the details on which the summary trees are to be built.
table	The underlying PeopleSoft data format, in which data is stored by columns (fields) and rows (records, or instances).

TableSet sharing	Specifies control table data for each business unit so that redundancy is eliminated.
target currency	The value of the entry currency or currencies converted to a single currency for budget viewing and inquiry purposes.
template	A template is HTML code associated with a Web page. It defines the layout of the page and also where to get HTML for each part of the page. In PeopleSoft, you use templates to build a page by combining HTML from a number of sources. For a PeopleSoft portal, all templates must be registered in the portal registry, and each content reference must be assigned a template.
territory	In PeopleSoft Sales Incentive Management, hierarchical relationships of business objects, including regions, products, customers, industries, and participants.
TimeSpan	A relative period, such as year-to-date or current period, that can be used in various PeopleSoft General Ledger functions and reports when a rolling time frame, rather than a specific date, is required. TimeSpans can also be used with flexible formulas in PeopleSoft Projects.
transaction allocation	In PeopleSoft Enterprise Incentive Management, the process of identifying the owner of a transaction. When a raw transaction from a batch is allocated to a plan context, the transaction is duplicated in the PeopleSoft Enterprise Incentive Management transaction tables.
transaction loading process	In PeopleSoft Enterprise Incentive Management, the process during which transactions are loaded into Sales Incentive Management. During loading, the source currency is converted to the business unit currency while retaining the source currency code. At the completion of this stage, the transaction is in the first state.
transaction state	In PeopleSoft Enterprise Incentive Management, a value assigned by an incentive rule to a transaction. Transaction states enable sections to process only transactions that are at a specific stage in system processing. After being successfully processed, transactions may be promoted to the next transaction state and picked up by a different section for further processing.
transaction type	In PeopleSoft Enterprise Incentive Management, a way to categorize transactions to identify specific transaction types (for example, shipment, order, opportunity, and so on). Plan sections process only one type of transaction type. Transaction types can be defined based on a company's specific processes model.
Translate table	A system edit table that stores codes and translate values for the miscellaneous fields in the database that do not warrant individual edit tables of their own.
tree	The graphical hierarchy in PeopleSoft systems that displays the relationship between all accounting units (for example, corporate divisions, projects, reporting groups, account numbers) and determines roll-up hierarchies.
unclaimed transaction	In PeopleSoft Enterprise Incentive Management, a transaction that is not claimed by a node or participant after the allocation process has completed, usually due to missing or incomplete data. Unclaimed transactions may be manually assigned to the appropriate node or participant by a compensation administrator.
uniform resource locator (URL)	In PeopleSoft, the term URL refers to the entire query string. The following is an example of a URL: http://serverx/InternetClient /InternetClientServlet?ICType=Script&ICScriptProgramName=WEBLIB_BEN_ 401k.PAGES.FieldFormula.iScript_Home401k
universal navigation header	Every PeopleSoft portal includes the universal navigation header, intended to appear at the top of every page as long as the user is signed on to the portal. In addition to providing access to the standard navigation buttons (like Home, Favorites, and signoff) the universal navigation header can also display a welcome message for each user.

URL	See *uniform resource locator (URL).*
user interaction object	In PeopleSoft Sales Incentive Management, used to define the reporting components and reports that a participant can access in his or her context. All Sales Incentive Management user interface objects and reports are registered as user interaction objects. User interaction objects can be linked to a compensation structure node through a compensation relationship object (individually or as groups).
variable	In PeopleSoft Sales Incentive Management, the intermediate results of calculations. Variables hold the calculation results and are then inputs to other calculations. Variables can be plan variables that persist beyond the run of an engine or local variables that exist only during the processing of a section.
warehouse	A PeopleSoft data warehouse that consists of predefined ETL maps, data warehouse tools, and DataMart definitions.
worksheet	A way of presenting data through a PeopleSoft Business Analysis Modeler interface that enables users to do in-depth analysis using pivoting tables, charts, notes, and history information.
workflow	The background process that creates a list of administrative actions based on selection criteria and specifies the procedure associated with each action.
worklist	The automated to-do list that PeopleSoft Workflow creates. From the worklist, you can directly access the pages you need to perform the next action, and then return to the worklist for another item.
zero-rated VAT	Abbreviation for *zero-rated value-added tax.* A VAT transaction with a VAT code that has a tax percent of zero. Used to track taxable VAT activity where no actual VAT amount is charged.

Index

A

Abs function 27
%Abs meta-SQL function 518
accessing
  AESection 592
  Component Interface structure 812
  PSMessages collection 1570
AccruableDays function 27
AccrualFactor function 28
Acos function 29
ActiveRowCount function 29
AddAttachment function 31
AddDropDownItem function 845
AddEmailAddress function 35
additional documentation xc
AddKeyListItem function 36
AddSystemPauseTimes function 37
AddToDate function 39
AddToDateTime function 40
AddToTime function 40
AESection class
  accessing 592
  declaring 595
  example 593
  GetAESection function 233
  methods 595
  property 600
  scope of 595
All function 41
AllOrNone function 42
AllowEmplIdChg function 43
%AllowNotification variable 563
%AllowRecipientLookup variable 563
Amortize function 44
API functions
  GetSession 269
  RemoteCall 364
  SendMail 394
application classes
  base classes 624
  Class function 627
  class names 603
  CollectGarbage function 70
  constants 615
  constructors 608
  declaring 625
  declaring external functions 610
  declaring external interfaces 604
  declaring private 605
  defining methods 606
  downcasting 620
  exception handling 621
  extending 604
  Get function 628
  global variables 619
  import declarations 612
  Import function 629
  Method function 629
  naming standards 611
  overriding properties 616
  overriding variables and properties 619
  private instance variables 619
  properties 615
  scope of 626
  self-reference 613
  Set function 630
  structure 602
  superclass reference 619
  understanding 601
  using 602
  variables 618
Application Engine
  CallAppEngine function 58
  CommitWork function 71
  file class example 911
  GetAESection function 233
  SetTempTableInstance function 431
  SQL object considerations 1621
  using with Business Interlink 666
application fundamentals lxxxix
%Appserver meta-HTML 585
%AppsRel meta-HTML 585
Array class
  CreateArray function 92
  CreateArrayAny function 93
  CreateArrayRept function 94
  creating arrays 633
  creating empty arrays 637
  declaring 640
  fetch results 1617

flattening and promotion 640
methods 641
multi-dimensional arrays 637
populating an array 635
properties 656
removing items 636
scope of 640
Split function 441
SQL bind values 1617
Array functions
CreateArray 92
CreateArrayAny 93
CreateArrayRept 94
Split 441
Asin function 45
%AsOfDate variable 563
AssignedPagelet
class properties 1195
collection methods 1197
collection property 1200
Atan function 46
attachment functions
AddAttachment 31
CopyAttachments 88
DeleteAttachment 135
filenames for PutAttachment 354
GetAttachment 234
PutAttachment 353
ViewAttachment 494
attachments
mobile 1640
OnGetProperty event 1641
OnGetPropertyFilter event 1642
OnSetProperty event 1642
AttributeValue
class properties 1176
collection methods 1178
collection property 1181
%AuthenticationToken variable 564
AvailableCategory
collection methods 1201
collection property 1202
AvailablePagelet
class properties 1203
collection methods 1205
collection property 1206

B

base application classes 624
%BINARYSORT meta-SQL function 518
BlackScholesCall function 46
BlackScholesPut function 47
BootstrapYTMs function 48
%BPName variable 564
branch collection
method 1666
properties 1667
Break 49
%Browser meta-HTML 585
%BrowserPlatform meta-HTML 585
%BrowserVersion meta-HTML 585
bulk functions
BulkDeleteField 49
BulkInsertField 51
BulkModifyPageFieldOrder 54
BulkUpdateIndexes 56
BulkDeleteField function 49
BulkInsertField function 51
BulkModifyPageFieldOrder function 54
BulkUpdateIndexes function 56
Business Interlink class
configuration parameters 722
declaring 666
GetBiDoc function 236
GetInterlink function 250
Incoming Business Interlink methods 708
Incoming Business Interlink properties 717
%MaxInterlinkSize variable 570
methods 667
property 721
scope of 667
state of object 666
using 660
using with Application Engine 666

C

CallAppEngine function 58
Canceling functions
DoCancel 147
Exit 176
WinEscape 502
CancelPubHeaderXmlDoc function 61
CancelPubXmlDoc function 61
CancelSubXmlDoc function 62
catch section 837
ChangeEmailAddress function 63

Char function 64
Chart class
  appearance properties 748
  axes methods 755
  axes properties 756
  color methods 753
  component processor 736
  creating a chart 766
  creating a Gantt chart 768
  data methods 739
  data properties 747
  data type 738
  examples 766
  Gantt chart 768
  GetChart function 239
  GetChartURL function 240
  iScript example 770
  iScripts 737
  legend method 761
  legend properties 762
  overlay methods 763
  overlay properties 765
  scope of 738
  style sheets 735
  terms 727
  title properties 759
  types 728
  understanding 725
Chart class functions
  GetChart 239
  GetChartURL 240
ChartField functions
  RenameDBField 368
  RenamePage 370
  RenameRecord 371
  SetDBFieldAuxFlag 405
  SetDBFieldCharDefn 407
  SetDBFieldFormat 408
  SetDBFieldFormatLength 410
  SetDBFieldLabel 411
  SetDBFieldLength 412
  SetDBFieldNotUsed 414
  SetPageFieldPageFieldName 423
  SetRecFieldEditTable 425
  SetRecFieldKey 427
CharType function 64
Chat functions
  InitChat 293
ChDir function 66
ChDrive function 66
cheat sheets 1813
CI collection
  mobile method 1042
  mobile properties 1043
CI data collection
  methods 805
  standard properties 811
class constructors
  application classes 608
  notifications classes 1063
Class function 627
Clean function 67
ClearDropDownList function 846
ClearKeyList function 67
ClearSearchDefault function 68
ClearSearchEdit function 69
%ClientDate variable 564
%ClientTimeZone variable 564
%COALESCE meta-SQL function 519
Code function 69
Codeb function 70
CollectGarbage function 70
%Cols meta-HTML 586
comments, submitting xciii
CommitWork function 71
common elements xciii
CompareLikeFields function 73
%CompIntfcName variable 565
CompIntfPropInfoCollection collection methods 814
CompIntfPropInfoCollection collection properties 815
CompIntfPropInfoCollection object properties 816
component buffer functions
  ActiveRowCount 29
  AddKeyListItem 36
  CompareLikeFields 73
  ComponentChanged 75
  CopyFields 89
  CopyRow 90
  CurrentLevelNumber 116
  CurrentRowNumber 116
  DeleteRecord 139
  DeleteRow 140
  DiscardRow 146
  ExpandBindVar 177
  FetchValue 181
  FieldChanged 182
  GetNextNumber 257

GetNextNumberWithGaps 259
GetRelField 266
GetSetId 270
InsertRow 297
PriorValue 349
RecordChanged 358
RecordDeleted 359
RecordNew 361
RowFlush 383
RowScrollSelect 384
RowScrollSelectNew 386
ScrollFlush 388
ScrollSelect 389
ScrollSelectNew 392
SetControlValue 402
SetCursorPos 404
SetDefault 415
SetDefaultAll 416
SetDefaultNext 417
SetDefaultNextRel 417
SetDefaultPrior 418
SetDefaultPriorRel 418
SetNextPage 422
SetTempTableInstance 431
SortScroll 440
StopFetching 449
TotalRowCount 462
TreeDetailInNode 480
UpDateSysVersion 490
UpdateValue 490
component buffer meta-SQL functions
%TruncateTable 557
component buffer system variables
%Component 564
%Menu 571
%Mode 571
Component Interface
mobile pages 1010
Component Interface class
accessing structure 812
%CompIntfcName variable 565
CompIntfPropInfoCollection collection methods 814
CompIntfPropInfoCollection collection properties 815
CompIntfPropInfoCollection object properties 816
Component Interface collection methods 785
Component Interface collection property 787
data collection methods 805
data collection standard properties 811
data collections 780
data item standard property 811
declaring 778
effective dated data 782
GetMethodNames function 254
implementing 779
keys 776
life cycle 774
quick reference 1862
reusing existing code 783
scope of 779
session class methods 784
setting keys 776
standard methods 787
standard properties 777, 801
timeouts 780
traversing 780
using data collection 780
Component Interface class function
StartWork 447
%Component meta-HTML 586
Component statement 74
%Component variable 564
ComponentChanged function 75
%Concat meta-SQL function 519
Consolidated Publications Incorporated (CPI) xc
constructors 608
contact information xciii
ContainsCharType function 75
ContainsOnlyCharType function 77
content reference
class methods 1161
class properties 1162
collection methods 1172
collection property 1175
%ContentID variable 565
%ContentType variable 565
Conversion functions
Char 64
Code 69
Codeb 70
ConvertChar 79
NumberToString 337
String 451
Value 493

ConvertChar function 79
ConvertCurrency function 83
ConvertDatetimeToBase function 85
ConvertRate function 86
ConvertTimeToBase function 87
Cookie class properties 956
CopyAttachments function 88
CopyFields function 89
%Copyright meta-HTML 586
%Copyright variable 566
CopyRow function 90
Cos function 91
Cot function 92
country-specific documentation xcii
CreateArray function 92
CreateArrayAny function 93
CreateArrayRept function 94
CreateDirectory function 96
CreateException function 97
CreateJavaArray function 98
CreateJavaObject function 99
CreateMessage function 100
CreateObject function 101
CreateObjectArray function 103
CreateProcessRequest function 104
CreateRecord function 105
CreateRowset function 107
CreateSOAPDoc function 109
CreateSQL function 109
CreateXmlDoc function 111
creating
  arrays 633
  web libraries 933
cross-references xcii
CubicSpline function 112
CurrEffDt function 114
CurrEffRowNum function 114
CurrEffSeq function 115
Currency and financial functions
  Amortize 44
  ConvertCurrency 83
  %Currency 566
  RoundCurrency 382
  SinglePaymentPV 439
  UniformSeriesPV 489
%Currency variable 566
Current date and time functions
  %CurrentDateIn 520
  %CurrentDateOut 520
  %CurrentDateTimeIn 520
  %CurrentDateTimeOut 520
  %CurrentTimeIn 521
  %CurrentTimeOut 521
%CurrentDateIn meta-SQL function 520
%CurrentDateOut meta-SQL function 520
%CurrentDateTimeIn meta-SQL function 520
%CurrentDateTimeOut meta-SQL function 520
CurrentLevelNumber function 116
CurrentRowNumber function 116
%CurrentTimeIn meta-SQL function 521
%CurrentTimeOut meta-SQL function 521
Custom display format functions
  GetStoredFormat 273
  SetDisplayFormat 419
Customer Connection Website xc

D

Data Buffer Access Functions
  CreateRecord 105
  CreateRowset 107
  GetField 242
  GetLevel0 253
  GetRecord 265
  GetRow 267
  GetRowset 268
data collection
  mobile methods 1044
  mobile properties 1048
Date and time functions
  AddToDate 39
  AddToDateTime 40
  AddToTime 40
  ConvertDatetimeToBase 85
  ConvertTimeToBase 87
  Date 117
  Date3 118
  %DateAdd 521
  %DateDiff 521
  %DateNull 522
  %DateOut 523
  DatePart 118
  %DatePart 523
  DateTime6 119
  %DateTimeDiff 523
  %DateTimeIn 524
  %DateTimeNull 524

%DateTimeOut 525
DateTimeToHTTP 120
DateTimeToLocalizedString 121
DateTimeToTimeZone 124
DateTimeValue 125
DateValue 126
Day 127
Days 128
Days360 128
Days365 129
%DTTM 526
FormatDateTime 192
GetCalendarDate 237
Hour 291
IsDate 301
IsDateTime 302
IsDaylightSavings 302
IsTime 311
meta-SQL considerations 512
Minute 326
Month 327
Second 393
Time 458
Time3 458
%TimeAdd 553
%TimeIn 554
%TimeNull 554
%TimeOut 555
TimePart 459
%TimePart 555
TimeToTimeZone 460
TimeValue 461
TimeZoneOffset 461
Weekday 501
Year 509
Date function 117
%Date variable 566
Date3 function 118
%DateAdd meta-SQL function 521
%DateDiff meta-SQL function 521
%DateNull meta-SQL function 522
%DateOut meta-SQL function 523
DatePart function 118
%DatePart meta-SQL function 523
%DateTime variable 567
DateTime6 function 119
%DateTimeDiff meta-SQL function 523
%DateTimeIn meta-SQL function 524
%DateTimeNull meta-SQL function 524
%DateTimeOut meta-SQL function 525
DateTimeToHTTP function 120
DateTimeToLocalizedString function 121
DateTimeToTimeZone function 124
DateTimeValue function 125
DateValue function 126
Day function 127
Days function 128
Days360 function 128
Days365 function 129
DBCSTrim function 129
%DBName meta-HTML 586
%DbName variable 567
DBPatternMatch function 129
%DbServerName variable 567
%DBType meta-HTML 586
%DbType variable 567
debugging
mobile 1017
%DecDiv meta-SQL function 525
Declare Function 130
%DecMult meta-SQL function 526
Decrypt function 133
Default functions
SetDefault 415
SetDefaultAll 416
SetDefaultNext 417
SetDefaultNextRel 417
SetDefaultPrior 418
SetDefaultPriorRel 418
Degrees function 134
DeleteAttachment function 135
DeleteEmailAddress function 137
DeleteImage function 138
DeleteRecord function 139
DeleteRow function 140
DeleteSQL function 141
DeleteSystemPauseTimes function 143
deprecated functions 1933
DeQueue function 144
%DeviceType system variable 567
DisableMenuItem function 146
DiscardRow function 146
Distribution List
determining for PostReport 1261
DoCancel function 147
documentation
country-specific xcii
printed xc
related xc
updates xc

DoModal function 148
DoModalComponent function 150
DoSave function 154
DoSaveNow function 155
Double-byte character functions
  CharType 64
  ContainsCharType 75
  ContainsOnlyCharType 77
  ConvertChar 79
  DBCSTrim 129
downcasting 620
%DTTM meta-SQL function 526
DynamicCategory collection
  methods 1207
  property 1209

E

%EffDtCheck meta-SQL function 527
Effective date functions
  CurrEffDt 114
  CurrEffRowNum 114
  CurrEffSeq 115
  %EffDtCheck 527
  NextEffDt 331
  NextRelEffDt 331
  PriorEffDt 347
  PriorRelEffDt 348
email functions
  AddEmailAddress 35
  ChangeEmailAddress 63
  DeleteEmailAddress 137
  MarkPrimaryEmailAddress 317
%EmailAddress variable 568
%EmployeeId variable 568
EnableMenuItem function 157
%Encode meta-HTML 587
EncodeURL function 157
EncodeURLForQueryString function 158
Encrypt function 159
EncryptNodePswd function 160
EndMessage function 160
EndModal function 161
EndModalComponent function 162
EnQueue function 163
environment functions
  ExpandEnvVar 178
  GetCwd 241
  GetEnv 241
Error 165
EscapeHTML function 167
EscapeJavascriptString function 168
EscapeWML function 169
Evaluate 169
Exact function 170
exception class
  CreateException function 97
  data type 838
  methods 839
  properties 841
  scope of 838
  Throw language construct 457
  throwing exceptions 836
  Try language construct 482
  try-catch sections 837
  understanding 835
  using 836
exception classes
  application classes 621
Exec function 171
Executable files
  Exec function 171
  WinExec function 503
ExecuteRolePeopleCode function 173
ExecuteRoleQuery function 174
ExecuteRoleWorkflowQuery
 function 175
Existence functions
  All 41
  AllOrNone 42
  None 335
  OnlyOne 344
  OnlyOneOrNone 345
Exit 176
exiting loops 49
Exp function 176
ExpandBindVar function 177
ExpandEnvVar function 178
ExpandSqlBinds function 178
%ExternalAuthInfo variable 568

F

Fact function 179
faults
  SOAPDoc class 1593
favorite
  class properties 1238
  collection methods 1239
  collection property 1242
FetchSQL function 180
FetchValue function 181

field class
declaring 844
GetField function 242
methods 844
properties 852
scope of 844
shortcut considerations 843
FieldChanged function 182
file attachment
AddAttachment function 31
CopyAttachments function 88
DeleteAttachment function 135
GetAttachment function 234
PutAttachment filenames 354
PutAttachment function 353
ViewAttachment function 494
file class
access interruptions 875
Application Engine example 911
CreateDirectory function 96
error processing 878
examples 901
file rowset considerations 911
FileExists function 184
FindFiles function 189
GetFile function 243
methods 879
multiple file layouts 915
plain text files 876
properties 898
reading multiple file layouts 915
ReadRecord example 905
ReadRowset example 910
RemoveDirectory function 367
scope of 874
security considerations 875
WriteRecord example 902
WriteRowset example 906
writing multiple file layouts 917
File functions
ChDir 66
ChDrive 66
FileExists function 184
Financial functions
AccruableDays 27
AccrualFactor 28
BlackScholesCall 46
BlackScholesPut 47
BootstrapYTMs 48
ConvertRate 86
CubicSpline 112
HermiteCubic 285
HistVolatility 290
LinearInterp 314
Find function 185
Findb function 186
FindCodeSetValues function 187
FindFiles function 189
flattening arrays 640
FlushBulkInserts function 191
folder
class method 1151
class properties 1152
collection methods 1157
collection property 1160
folder management
determining Distribution List for PostReport 1261
PostReport class methods 1262
PostReport class overview 1261
PostReport class properties 1264
For 192
FormatDateTime function 192
%Formname meta-HTML 587
Forward function 193
Function statement 196
functions
declaring 130
declaring for application classes 610
defining 196
return values 376

G

Gantt chart example 768
GenerateActGuideContentUrl function 198
GenerateActGuidePortalUrl function 199
GenerateActGuideRelativeUrl function 200
GenerateComponentContentRelURL function 201
GenerateComponentContentURL function 203
GenerateComponentPortalRelURL function 205
GenerateComponentPortalURL function 207
GenerateComponentRelativeURL function 209
GenerateExternalPortalURL function 211

GenerateExternalRelativeURL function 212
GenerateHomepagePortalURL function 213
GenerateHomepageRelativeURL function 214
GenerateMobileTree function 215
GenerateQueryContentURL function 218
GenerateQueryPortalURL function 220
GenerateQueryRelativeURL function 221
GenerateScriptContentRelURL function 223
GenerateScriptContentURL function 224
GenerateScriptPortalRelURL function 226
GenerateScriptPortalURL function 227
GenerateScriptRelativeURL function 229
GenerateTree function 230
GenerateWorklistPortalURL function 231
GenerateWorklistRelativeURL function 232
Get function 628
GetAESection function 233
GetAttachment function 234
GetBiDoc function 236
GetCalendarDate function 237
GetChart function 239
GetChartURL function 240
GetCwd function 241
GetEnv function 241
GetField function 242
GetFile function 243
GetGrid function 246
GetHTMLText function 248
GetImageExtents function 249
GetInterlink function 250
GetJavaClass function 252
GetLevel0 function 253
GetMessage function 255
GetMessageInstance function 255
GetMessageXmlDoc function 255
GetNextNumber function 257
GetNextNumberWithGaps function 259
GetNextProcessInstance function 261
GetNRXmlDoc function 261
GetPage function 262
GetPubContractInstance function 263
GetPubHeaderXmlDoc function 263
GetPubXmlDoc function 264
GetRecord function 265
GetRelField function 266
GetRow function 267
GetRowset function 268
GetSession function 269
GetSetId function 270
GetSQL function 271
GetStoredFormat function 273
GetSubContractInstance function 274
GetSubXmlDoc function 276
GetSyncLogData function 277
GetURL function 279
GetUserOption function 280
GetWLFieldValue function 281
Global statement 281
glossary 1949
Gray function 282
grid class
  changing a grid name 247
  declaring 923
  GetGrid function 246
  gridcolumn 926
  method 924
  properties 926
  scope of 923
  shortcut considerations 922
  specifying grid column name 924
  using in PeopleCode 922
gridcolumn class
  properties 927

H

Hash function 284
having criteria 1316
HermiteCubic function 285
Hide function 285
HideMenuItem function 287
HideRow function 287
HideScroll function 289
HistVolatility function 290
Hour function 291
%HPTabName variable 568
HTML
  EscapeHTML function 167
  GetHTMLText function 248
HTML variables
  %Appserver 585
  %AppsREl 585
  %Browser 585
  %BrowserPlatform 585
  %BrowserVersion 585

%Cols 586
%Component 586
%Copyright 586
%DBName 586
%DBType 586
%Encode 587
%Formname 587
%Image 587
%JavaScript 587
%LabelTag 588
%Menu 588
%Message 588
%Page 589
%ServicePack 589
%SubmitScriptName 589
%ToolsRel 590
%URL 590
%UserId 590

I

Idiv function 291
If 292
image functions
DeleteImage 138
GetImageExtents 249
InsertImage 295
%Image meta-HTML 587
implementing
Component Interface objects 779
tree classes 1660
Tree Viewer 1942
import declarations 612
Import function 629
Import manager
%Import system variable 568
%Import variable 568
importing
application classes 612
notification classes 1062
Incoming Business Interlink methods 708
Incoming Business Interlink properties 717
InitChat function 293
InsertImage function 295
InsertRow function 297
%InsertSelect meta-SQL function 529
%InsertValues meta-SQL function 532
Int function 298
Integer function 299
Integration Broker functions
EncryptNodePswd 160
FindCodeSetValues 187
NodeDelete 332
NodeRename 333
NodeSaveAs 333
NodeTranDelete 334
RelNodeTranDelete 363
Transform 476
TransformEx 478
internet functions
AddEmailAddress 35
ChangeEmailAddress 63
%ContentID variable 565
%ContentType variable 565
CreateSOAPDoc 109
DeleteEmailAddress 137
%EmailAddress variable 568
EncodeURL 157
EncodeURLForQueryString 158
EscapeHTML 167
EscapeJavascriptString 168
EscapeWML 169
GenerateActGuideContentUrl 198
GenerateActGuidePortalUrl 199
GenerateActGuideRelativeUrl 200
GenerateComponentContentRelURL 201
GenerateComponentContentURL 203
GenerateComponentPortalRelURL 205
GenerateComponentPortalURL 207
GenerateComponentRelativeURL 209
GenerateExternalPortalURL 211
GenerateExternalRelativeURL 212
GenerateHomepagePortalURL 213
GenerateHomepageRelativeURL 214
GenerateQueryContentURL 218
GenerateQueryPortalURL 220
GenerateQueryRelativeURL 221
GenerateScriptContentRelURL 223
GenerateScriptContentURL 224
GenerateScriptPortalRelURL 226
GenerateScriptPortalURL 227
GenerateScriptRelativeURL 229
GenerateTree 230
GenerateWorklistPortalURL 231
GenerateWorklistRelativeURL 232
GetChartURL 240
GetHTMLText 248
GetMethodNames 254
GetURL 279
%HPTabName variable 568

%LocalNode variable 569
MarkPrimaryEmailAddress 317
%Node variable 571
%Portal variable 575
%Request variable 576
%Response variable 576
%RunningInPortal variable 577
Unencode 483
%UserDescription variable 581
ViewContentURL 498
ViewURL 498
Internet Script
chart example 770
charts 737
creating 933
Internet Script classes
accessing 936
Cookie class 941
Cookie class properties 956
creating web libraries 933
data types 939
generate functions 937
overview 931
quick reference 1924
Request class 940
Request class methods 942
Request class properties 944
Response class 940
Response class methods 950
scope of 939
security 934
URI 932
URL 932
when to use 935
IsAlpha Function 300
IsAlphaNumeric Function 300
iScript 931
See Also See Internet Script classes
IsDate Function 301
IsDateTime Function 302
IsDaylightSavings function 302
IsDigits Function 304
IsHidden function 304
IsMenuItemAuthorized function 305
IsMessageActive function 307
IsModal function 307
IsModalComponent function 308
%IsMultiLanguageEnabled variable 569
IsNumber Function 309
IsSearchDialog function 310
IsTime Function 311
IsUserInPermissionList function 312
IsUserInRole function 312

J

Java class
CreateJavaArray function 98
CreateJavaObject function 99
GetJavaClass function 252
%JavaScript meta-HTML 587
%Join meta-SQL function 533

K

%KeyEqual meta-SQL function 535
%KeyEqualNoEffDt meta-SQL function 536

L

%LabelTag meta-HTML 588
Language constructs
Break 49
Component 74
Declare function 130
Evaluate 169
Exit 176
For 192
Function 196
Global 281
If 292
Local 315
Repeat 372
Return 376
Throw 457
Try 482
While 502
Language preferences
%IsMultiLanguageEnabled variable 569
%Language variable 569
%Language_Base variable 569
SetLanguage function 420
%Language variable 569
%Language_Base variable 569
leaf class
methods 1668
properties 1674
Left function 313
Len function 313
Lenb function 314

level
class methods 1680
class properties 1681
collection methods 1678
collection properties 1679
%Like meta-SQL function 537
%LikeExact meta-SQL function 539
LinearInterp function 314
%List meta-SQL function 541
%ListBind meta-SQL function 544
%ListEqual meta-SQL function 544
ListViewAttrs
mobile method 1049
mobile properties 1050
Ln function 315
Local statement 315
%LocalNode variable 569
Log10 function 316
loops
exiting 49
Lower function 316
LTrim function 317

M

mapping
functions to methods and properties 1928
old names to new names 1931
%Market variable 570
MarkPrimaryEmailAddress function 317
MarkWLItemWorked function 318
Math functions
Abs 27
%Abs meta-SQL function 518
Acos 29
Asin 45
Atan 46
Cos 91
Cot 92
Degrees 134
IsNumber 309
Max 319
Min 326
Radians 356
Sin 439
Tan 456
Max Function 319
%MaxInterlinkSize variable 570
%MaxMessageSize variable 570
Menu functions
DisableMenuItem 146
EnableMenuItem 157
HideMenuItem 287
UnGray 484
%Menu meta-HTML 588
%Menu variable 571
message class
considerations with character fields 978
considerations with partial records 978
considerations with rowset 976
CreateMessage function 100
declaring 974
DeleteSystemPauseTimes function 143
error handling 979
GetMessage function 255
GetMessageInstance function 255
GetPubContractInstance function 263
GetSubContractInstance function 274
GetSyncLogData function 277
IsMessageActive function 307
methods 980
populating 975
properties 998
ReturnToServer function 377
scope of 974
Message display functions
EndMessage 160
Error 165
MessageBox 319
MsgGet 328
MsgGetExplainText 329
MsgGetText 330
Prompt 350
Warning 499
WinMessage 505
%Message meta-HTML 588
MessageBox function 319
messaging
AddSystemPauseTimes function 37
CancelPubHeaderXmlDoc 61
CancelPubXmlDoc 61
CancelSubXmlDoc 62
CreateMessage function 100
CreateXmlDoc function 111
DeleteSystemPauseTimes function 143
GetMessage function 255
GetMessageInstance function 255
GetMessageXmlDoc function 255
GetNRXmlDoc function 261

GetPubContractInstance function 263
GetPubHeaderXmlDoc 263
GetPubXmlDoc 264
GetSubContractInstance function 274
GetSubXmlDoc 276
GetSyncLogData function 277
IsMessageActive function 307
%MaxMessageSize variable 570
message class 973
PingNode function 346
PublishXmlDoc function 352
ReturnToServer function 377
SetChannelStatus function 401
SOAP 1588
SOAPDoc components 1588
SyncRequestXmlDoc function 455
Transform function 476
TransformEx function 478
using SOAPDoc objects 1594
meta-HTML
%Appserver 585
%AppsRel 585
%Browser 585
%BrowserPlatform 585
%BrowserVersion 585
%Cols 586
comments 584
%Component 586
%Copyright 586
%DBName 586
%DBType 586
%Encode 587
%Formname 587
functions 584
%Image 587
%JavaScript 587
%LabelTag 588
%Menu 588
%Message 588
%Page 589
%ServicePack 589
%SubmitScriptName 589
%ToolsRel 590
understanding 583
%URL 590
%UserId 590
variables 584
Meta-SQL functions 511
%Abs 518
%BINARYSORT 518
%COALESCE 519
%Concat 519
%CurrentDateIn 520
%CurrentDateOut 520
%CurrentDateTimeIn 520
%CurrentDateTimeOut 520
%CurrentTimeIn 521
%CurrentTimeOut 521
date considerations 512
%DateAdd 521
%DateDiff 521
%DateNull 522
%DateOut 523
%DatePart 523
%DateTimeDiff 523
%DateTimeIn 524
%DateTimeNull 524
%DateTimeOut 525
%DecDiv 525
%DecMult 526
%DTTM 526
%FirstRows 528
%OPRCLAUSE 547
placement considerations 513
%Round 548
shortcuts 560
%SUBREC 551
%Substring 550
%Test 552
%TimeIn 554
%TimeNull 554
%TimeOut 555
%TimePart 555
%TrimSubstr 556
%Truncate 556
%TruncateTable 557
Method function 629
methods
defining for application classes 606
Min Function 326
Minute function 326
Mobile Agent
advanced synchronization considerations 1649
built-in functions available 1057
Component Interface 1010
debugging mobile classes 1017
detail views 1011
HTML areas 1016
list views 1011

PeopleCode 1013
synchronization considerations 1635
synchronization server events 1633
SyncServer Class 1633
mobile class
CI collection method 1042
CI collection properties 1043
data collection methods 1044
data collection properties 1048
data types 1020
debugging 1017
%DeviceType system variable 567
error handling 1019
GenerateMobileTree function 215
hierarchy 1013
HTML areas 1016
ListViewAttrs method 1049
ListViewAttrs properties 1050
methods 1022
%MobilePage system variable 571
PeerDefaultAttributes method 1056
PeerDefaultAttributes properties 1057
property 1036
PropertyAttrs method 1052
PropertyAttrs properties 1053
PropertyInfo class properties 1039
PropertyInfo collection method 1037
PropertyInfo collection property 1038
scope of 1020
session class method 1021
synchronizable Component Interfaces 1010
%ThisMobileObject system variable 580
TransferMobilePage function 467
mobile classes
quick reference 1920
mobile events
OnConflict 1636
OnGetDefinition 1650
OnGetProperty 1641
OnGetPropertyFilter 1642
OnSelect 1637
OnSetProperty 1642
OnValidate 1639
OnValidateSet 1649
using attachment events 1640
Mobile PeopleCode, *See* PeopleCode
%MobilePage system variable 571
Mod function 327
Modal component functions
DoModalComponent 150
EndModalComponent 162
IsModalComponent 308
%Mode variable 571
Month function 327
MsgGet function 328
MsgGetExplainText function 329
MsgGetText function 330
MultiChannel Framework functions
DeQueue 144
EnQueue 163
Forward 193
InitChat 293
NotifyQ 336

N

%NavigatorHomePermissionList variable 571
NextEffDt function 331
NextRelEffDt function 331
node class (PortalRegistry API)
properties 1137
node class (Tree API)
methods 1683
properties 1695
%Node variable 571
NodeDelete function 332
NodeRename function 333
NodeSaveAs function 333
NodeTranDelete function 334
None function 335
notes xcii
Notification class
method 1066
properties 1067
notification classes
constructors 1063
creating 1062
creating a WorklistEntry 1084
examples 1084
importing 1062
notification class method 1066
notification class properties 1067
NotificationAddress class properties 1071
NotificationTemplate class methods 1072
NotificationTemplate class properties 1075

scope of 1061–1062
updating a WorklistEntry 1086
WorklistEntry class methods 1077
WorklistEntry class properties 1080
NotificationAddress
class properties 1071
NotificationTemplate
class methods 1072
class properties 1075
NotifyQ function 336
%NoUppercase meta-SQL function 545
NumberToString Function 337
Numeric functions
Abs 27
Acos 29
Asin 45
Atan 46
Cos 91
Cot 92
%DecDiv 525
%DecMult 526
Degrees 134
Exp 176
Fact 179
Idiv 291
Int 298
Integer 299
Ln 315
Log10 316
Mod 327
Product 349
Radians 356
Rand 357
Round 381
%Round 548
Sign 438
Sin 439
Sqrt 447
Tan 456
Truncate 481
%Truncate 556
%NumToChar meta-SQL function 546

O

object functions
CreateObject 101
CreateObjectArray 103
ObjectDoMethod 340
ObjectDoMethodArray 341
ObjectGetProperty 342
ObjectSetProperty 343
ObjectDoMethod function 340
ObjectDoMethodArray function 341
ObjectGetProperty function 342
ObjectSetProperty function 343
%OldKeyEqual meta-SQL function 547
OnConflict mobile event 1636
OnGetDefinition mobile event 1650
OnGetProperty mobile event 1641
OnGetPropertyFilter mobile event 1642
OnlyOne function 344
OnlyOneOrNone function 345
OnSelect mobile event 1637
OnSetProperty mobile event 1642
OnValidate mobile event 1639
OnValidateSet mobile event 1649
%OPRCLAUSE meta-SQL function 547

P

page appearance functions
AddDropDownItem 845
ClearDropDownList 846
GetPage 262
Gray 282
Hide 285
HideRow 287
HideScroll 289
IsHidden 304
SetLabel 419
Unhide 486
UnhideRow 487
UnhideScroll 488
page class
data type 1090
GetPage function 262
properties 1090
scope of 1090
shortcut considerations 1089
%Page meta-HTML 589
%Page variable 573
pagelet
class properties 1218
collection methods 1224
collection property 1227
PageletCategory
class method 1210
class properties 1211
collection methods 1214
collection property 1217
ParseResult

class properties 1522
collection methods 1520
collection properties 1521
%PasswordExpired variable 574
PeerDefaultAttributes
method 1056
properties 1057
PeopleBooks
ordering xc
PeopleCode 1636
mobile events 1636
PeopleCode classes
AESection 591
application 601
array 633
business interlink 659
chart 725
Component Interface 773
exception 835
field 843
file 873
grid 921
gridcolumn 926
Internet Script 931
message 973
Mobile 1009
notification 1061
page 1089
PortalRegistry 1093
PostReport 1261
ProcessRequest 1269
query 1303
quick reference 1813
record 1441
row 1465
rowset 1479
search 1509
session 1559
SOAPDoc 1587
SQL 1613
SyncServer 1633
tree 1653
XmlDoc 1757
PeopleCode syntax
quick reference 1934
PeopleCode, mobile, *See* PeopleCode
PeopleCode, typographical conventions xci
PeopleSoft application fundamentals lxxxix
%PerfTime system variable 574
%PermissionLists variable 575
PermissionValue
class properties 1181
collection methods 1183
collection property 1185
personalization functions
GetUserOption 280
SetUserOption 437
PingNode function 346
portal
class method 1144
class properties 1144
collection methods 1145
collection properties 1147
portal registry classes
AssignedPagelet class properties 1195
AssignedPagelet collection methods 1197
AssignedPagelet collection property 1200
AttributeValue collection methods 1178
AttributeValue collection property 1181
AttributeValue properties 1176
AvailableCategory collection methods 1201
AvailableCategory collection property 1202
AvailablePagelet class properties 1203
AvailablePagelet collection methods 1205
AvailablePagelet collection property 1206
content reference collection methods 1172
content reference collection property 1175
content reference methods 1161
content reference properties 1162
content references 1106
data type 1112
deleting content considerations 1111
DynamicCategory collection methods 1207
DynamicCategory collection property 1209
error handling 1100
examples 1242

favorite class properties 1238
favorite collection methods 1239
favorite collection property 1242
folder collection methods 1157
folder collection property 1160
folder method 1151
folder properties 1152
hierarchy 1104
life-cycle 1101
methods 1117
node class properties 1137
node collection methods 1140
node collection property 1141
node template collection properties 1150
NodeTemplate class methods 1147
NodeTemplate collection methods 1148
overview 1093
pagelet collection methods 1224
pagelet collection property 1227
pagelet properties 1218
PageletCategory collection methods 1214
PageletCategory collection property 1217
PageletCategory method 1210
PageletCategory properties 1211
PermissionValue collection methods 1183
PermissionValue collection property 1185
PermissionValue properties 1181
portal class method 1144
portal class properties 1144
portal collection methods 1145
portal collection properties 1147
portal registry collection methods 1135
portal registry collection property 1136
properties 1132
quick reference 1873
RemoteNode collection methods 1142
RemoteNode collection property 1143
saving content considerations 1112
scope of 1113
search class methods 1515
security considerations 1097
SelectedPagelet class properties 1233
SelectedPagelet collection methods 1235
SelectedPagelet collection property 1237
session class methods 1113
TabDefinition collection methods 1192
TabDefinition collection property 1195
TabDefinition properties 1187
UserHomepage Class method 1227
UserHomepage Class properties 1228
UserTab Class properties 1229
UserTab collection methods 1230
UserTab collection property 1233
using 1096
using Cascading Permissions 1098
using PermissionValue 1098
ValidFrom and ValidTo 1100
%Portal variable 575
PostReport class
declaring 1262
determining Distribution List 1261
examples 1267
methods 1262
overview 1261
properties 1264
scope of 1262
SetPostReport function 425
prerequisites lxxxix
%PrimaryPermissionList variable 575
printed documentation xc
PriorEffDt function 347
PriorRelEffDt function 348
PriorValue function 349
private
instance variables 605
methods 605
private instance variables 619
Process Scheduler functions
CreateProcessRequest 104
GetNextProcessInstance 261
SetPostReport 425
SetupScheduleDefnItem 437
%ProcessProfilePermissionList variable 576
ProcessRequest class
CreateProcessRequest function 104
data type 1270
determining Distribution List for PostReport 1261
examples 1300
format values 1274
GetNextProcessInstance function 261

item options 1270
methods 1278
output type values 1274
PostReport class methods 1262
PostReport class overview 1261
PostReport class properties 1264
properties 1291
scope of 1270
SetupScheduleDefnItem function 437
Product function 349
promotion in arrays 640
Prompt function 350
Proper function 352
property overrides 616
PropertyAttrs
mobile method 1052
mobile properties 1053
PropertyInfo class
mobile properties 1039
PropertyInfo collection
mobile method 1037
mobile property 1038
%PSAuthResult variable 576
PSMessage class
properties 1573
PSMessages collection
accessing 1570
methods 1570
PublishXmlDoc function 352
PutAttachment function 353

Q

Query
class methods 1327
class properties 1339
collection methods 1324
collection property 1326
query classes 1303
adding criteria 1434
collections in 1303
creating a new query 1432
data type 1316
error handling 1314
examples 1432
having criteria 1316
hierarchy 1305
life-cycle 1304
overview 1307
Query class methods 1327
query class properties 1339
query collection 1324
Query collection methods 1324
query collection property 1326
query fields 1315
query metadata class properties 1408
query metadata collection methods 1406
query metadata collection property 1408
QueryCriteria class methods 1379
QueryCriteria class properties 1383
QueryCriteria collection methods 1376
QueryCriteria collection property 1378
QueryDBRecord class methods 1417
QueryDBRecord class properties 1418
QueryDBRecord collection methods 1414
QueryDBRecord collection property 1416
QueryDBRecordField class properties 1422
QueryDBRecordField collection methods 1419
QueryDBRecordField collection property 1422
QueryExpression class properties 1396
QueryExpression collection methods 1393
QueryExpression collection property 1395
QueryField class properties 1370
QueryField collection methods 1366
QueryField collection property 1368
QueryField methods 1369
QueryList methods 1399
QueryList property 1401
QueryListValue properties 1401
QueryOutputFields 1315
QueryPrompt class properties 1429
QueryPrompt collection methods 1426
QueryPrompt collection property 1428
QueryRecord class method 1363
QueryRecord class properties 1364
QueryRecord collection methods 1360
QueryRecord collection property 1362
QueryRecordHierarchy class properties 1405
QueryRecordHierarchy collection methods 1402

QueryRecordHierarchy collection property 1404
QuerySecurityProfile class properties 1411
QuerySelect class methods 1347
QuerySelect class properties 1354
QuerySelect collection methods 1357
QuerySelect collection property 1360
QuerySelectedFields 1315
QueryStatistics properties 1410
quick reference 1893
running a query 1313
scope of 1316
security 1314
session methods 1317
using query metadata 1311
working with criteria and expressions 1309
QueryCriteria
adding example 1434
class methods 1379
class properties 1383
collection methods 1376
collection property 1378
QueryDBRecord
class methods 1417
class properties 1418
collection methods 1414
collection property 1416
QueryDBRecordField
class properties 1422
collection methods 1419
collection property 1422
QueryExpression
class properties 1396
collection methods 1393
collection property 1395
QueryField
class methods 1369
class properties 1370
collection methods 1366
collection property 1368
QueryList
class methods 1399
class property 1401
QueryListValue
class properties 1401
QueryPrompt
class properties 1429
collection methods 1426
collection property 1428
QueryRecord
class method 1363
class properties 1364
collection methods 1360
collection property 1362
QueryRecordHierarchy
class properties 1405
collection methods 1402
collection property 1404
QuerySecurityProfile
class properties 1411
QuerySelect
class methods 1347
class properties 1354
collection methods 1357
collection property 1360
QueryStatistics
class properties 1410
Queue Server functions
DeQueue 144
EnQueue 163
Forward 193
NotifyQ 336
Quote function 356

R

Radians function 356
Rand function 357
record class
data type 1443
GetRecord function 265
methods 1443
properties 1460
scope of 1443
shortcut considerations 1441
SQL methods 1442
RecordChanged function 358
RecordDeleted function 359
RecordNew function 361
referencing
application classes 613
superclasses 619
regional settings object
about 1562
properties 1577
related documentation xc
RelNodeTranDelete function 363
RemoteCall
function 364

RemoveDirectory function 367
RenameDBField function 368
RenamePage function 370
RenameRecord function 371
Repeat 372
Replace function 373
Report functions
SetPostReport 425
Rept function 373
Request class
methods 942
properties 944
%Request variable 576
Response class methods 950
%Response variable 576
ReSubmitPubHeaderXmlDoc function 374
ReSubmitPubXmlDoc function 375
ReSubmitSubXmlDoc function 376
%ResultDocument variable 576
Return 376
ReturnToServer function 377
ReValidateNRXmlDoc function 379
RevalidatePassword function 379
Right function 381
%Roles variable 577
Round function 381
%Round meta-SQL function 548
RoundCurrency function 382
row class
data type 1466
GetRow function 267
methods 1466
properties 1471
scope of 1466
shortcut considerations 1466
RowFlush function 383
RowScrollSelect function 384
RowScrollSelectNew function 386
%RowSecurityPermissionList variable 577
rowset class
data type 1480
GetRowset function 268
methods 1481
properties 1502
scope of 1480
shortcut considerations 1480
RTrim function 387
Running executable files
Exec function 171
WinExec function 503
%RunningInPortal variable 577

S

Saving functions
DoSave 154
DoSaveNow 155
ScrollFlush function 388
ScrollSelect function 389
ScrollSelect functions
RowScrollSelect 384
RowScrollSelectNew 386
ScrollFlush 388
ScrollSelect 389
ScrollSelectNew 392
SortScroll 440
ScrollSelectNew function 392
search classes
data type 1514
error handling 1512
examples 1554
hierarchy 1512
overview 1509
ParseResult class properties 1522
ParseResult collection methods 1520
ParseResult collection properties 1521
PortalRegistry class methods 1515
quick reference 1866
scope of 1514
Search FS option class properties 1549
Search HTTP option class properties 1546
Search language collection methods 1539
Search start options class properties 1554
Search start options collection methods 1551
Search start options collection property 1553
SearchField class properties 1527
SearchField collection methods 1525
SearchField collection property 1526
SearchIndex class method 1529
SearchIndex class properties 1530
SearchIndexes collection methods 1527
SearchIndexes collection property 1529

SearchLanguage class properties 1542
SearchLanguage collection property 1541
SearchQuery class methods 1517
SearchQuery class properties 1519
SearchRecord option class properties 1532
SearchRecordField class properties 1538
SearchRecordField collection methods 1535
SearchRecordField collection property 1537
SearchResult class properties 1524
SearchResult collection methods 1522
SearchResult collection property 1524
SearchSchedule class properties 1545
SearchSchedule collection methods 1542
SearchSchedule collection property 1544
session class methods 1514
Search FS option class properties 1549
Search functions
AddKeyListItem 36
ClearKeyList 67
ClearSearchDefault 68
ClearSearchEdit 69
IsSearchDialog 310
SetSearchDefault 428
SetSearchDialogBehavior 429
SetSearchEdit 430
Search HTTP option class properties 1546
Search language
collection methods 1539
search query 1509
See Also See search classes
Search start options
class properties 1554
collection methods 1551
collection property 1553
SearchField
class properties 1527
collection methods 1525
collection property 1526
SearchIndex
class method 1529
class properties 1530
SearchIndexes
collection methods 1527
collection property 1529
SearchLanguage
class properties 1542
collection property 1541
SearchQuery
class methods 1517
class properties 1519
SearchRecord
option class properties 1532
SearchRecordField
class properties 1538
collection methods 1535
collection property 1537
SearchResult
class properties 1524
collection methods 1522
collection property 1524
understanding 1510
using collections 1510
SearchSchedule
class properties 1545
collection methods 1542
collection property 1544
Second function 393
Secondary page functions
DoModal 148
EndModal 161
IsModal 307
security
AllowEmplIdChg function 43
%AuthenticationToken variable 564
Decrypt function 133
%EmployeeId variable 568
Encrypt function 159
ExecuteRolePeopleCode function 173
ExecuteRoleQuery function 174
ExecuteRoleWorkflowQuery function 175
%ExternalAuthInfo variable 568
Hash function 284
Internet Scripts 934
IsMenuItemAuthorized function 305
IsUserInPermissionList function 312
IsUserInRole function 312
%NavigatorHomePermissionList variable 571
%OPRCLAUSE meta-SQL 547
%PasswordExpired variable 574

%PermissionLists variable 575
portal registry classes 1097
%PrimaryPermissionList variable 575
%ProcessProfilePermissionList variable 576
%PSAuthResult variable 576
query classes 1314
%ResultDocument variable 576
RevalidatePassword function 379
%Roles variable 577
%RowSecurityPermissionList variable 577
SetAuthenticationResult function 399
SetPasswordExpired function 424
%SignonUserId variable 578
%SignonUserPswd variable 578
SwitchUser function 454
%UserId variable 581
SelectedPagelet
class properties 1233
collection methods 1235
collection property 1237
self-reference 613
See Also See application classes, self-reference
SendMail function 394
%ServerTimeZone variable 577
%ServicePack meta-HTML 589
session class
access to PeopleSoft system 1560
Component Interface methods 784
declaring 1563
error handling 1560
GetSession function 269
methods 1564
mobil method 1021
portal registry methods 1113
properties 1567
PSMessages class properties 1573
PSMessages collection 1570
PSMessages collection methods 1570
query class methods 1317
quick reference 1859
regional settings overview 1562
regional settings properties 1577
scope of 1564
search method 1514
security 1560
trace settings overview 1562
trace settings properties 1580
tree classes methods 1663
%Session variable 577
Set function 630
SetAuthenticationResult function 399
SetChannelStatus function 401
SetControlValue function 402
SetCursorPos function 404
SetDBFieldAuxFlag function 405
SetDBFieldCharDefn function 407
SetDBFieldFormat function 408
SetDBFieldFormatLength function 410
SetDBFieldLabel function 411
SetDBFieldLength function 412
SetDBFieldNotUsed function 414
SetDefault function 415
SetDefaultAll function 416
SetDefaultNext function 417
SetDefaultNextRel function 417
SetDefaultPrior function 418
SetDefaultPriorRel function 418
SetLabel function 419
SetLanguage function 420
SetNextPage function 422
SetPageFieldPageFieldName function 423
SetPasswordExpired function 424
SetPostReport function 425
SetRecFieldEditTable function 425
SetRecFieldKey function 427
SetReEdit function 428
SetSearchDefault function 428
SetSearchDialogBehavior function 429
SetSearchEdit function 430
SetTempTableInstance function 431
SetTracePC function 432
SetTraceSQL function 435
SetupScheduleDefnItem function 437
SetUserOption function 437
shortcut considerations
field class 843
grid class 922
page class 1089
record class 1441
row class 1466
rowset class 1480
Sign function 438
%SignonUserId variable 578
%SignonUserPswd variable 578
Sin function 439
SinglePaymentPV function 439

%SMTPBlackberryReplyTo variable 578
%SMTPGuaranteed variable 578
%SMTPSender variable 579
SOAPDoc class
accessing sections 1592
body section 1591
CreateSOAPDoc function 109
creating 1590, 1608
data type 1594
envelope section 1591
examples 1608
fault section 1593
header section 1590
message exchange mode 1588
method section 1591
methods 1595
overview 1587
parts of 1588
properties 1605
reading 1608
required PeopleCode 1592
scope of 1594
using 1609
validating 1593
XmlDoc 1590
SortScroll function 440
Split function 441
SQL
binding 1615
CreateSQL function 109
DeleteSQL function 141
executing 1615
ExpandSqlBinds function 178
FetchSQL function 180
%FirstRows meta-SQL function 528
FlushBulkInserts function 191
GetSQL function 271
%InsertSelect meta-SQL function 529
%InsertValues meta-SQL function 532
%Join meta-SQL function 533
%KeyEqual meta-SQL function 535
%KeyEqualNoEffDt meta-SQL function 536
%Like meta-SQL function 537
%LikeExact meta-SQL function 539
%List meta-SQL function 541
%ListBind meta-SQL function 544
%ListEqual meta-SQL function 544
Meta functions 511
%NoUppercase meta-SQL function 545
%OldKeyEqual meta-SQL function 547
reusing a cursor 1618
SetTempTableInstance function 431
%SQL meta-SQL function 548
SQLExec function 442
%SQLRows variable 579
StoreSQL function 450
%Table meta-SQL function 551
%UpdatePairs meta-SQL function 558
SQL class
application engine considerations 1621
array of any 1617
binding and executing statements 1615
CreateSQL function 109
data type 1621
DeleteSQL function 141
fetching 1616
FetchSQL function 180
GetSQL function 271
global declaration considerations 1621
methods 1622
properties 1627
record methods 1615
reusing a cursor 1618
scope of 1621
SELECT styles 1617
SQL definition 1615
SQLExec function 442
StoreSQL function 450
SQL date and time functions
%DateOut 523
%DateTimeIn 524
%DateTimeOut 525
%DTTM 526
%TimeIn 554
%TimeOut 555
%TimePart 555
SQL definition
creating 1615
SQL functions
quote 356
%SQL meta-SQL function 548
SQL shortcuts
%Delete 560
%Insert 560
%SelectAll 560
%SelectByKey 562

%SelectByKeyEffDt 562
%SelectDistinct 561
%Update 562
SQLExec function 442
%SQLRows variable 579
Sqrt function 447
StartWork function 447
StopFetching function 449
StoreSQL function 450
String function 451
string functions
quote 356
String functions
Clean 67
Code 69
Codeb 70
%Concat 519
DBCSTrim 129
DBPatternMatch 129
Exact 170
ExpandBindVar 177
Find 185
Findb 186
IsAlpha 300
IsAlphaNumeric 300
IsDigits 304
Left 313
Len 313
Lenb 314
Lower 316
LTrim 317
NumberToString 337
%NumToChar 546
Proper 352
Replace 373
Rept 373
Right 381
RTrim 387
String 451
Substitute 452
%Substring 550
SubString 452
Substringb 453
%TextIn 553
%TrimSubstr 556
Upper 492
%Upper 559
Value 493
%SubmitScriptName meta-HTML 589
%SUBREC meta-SQL function 551
Subrecords
%SUBREC meta-SQL function 551
Substitute function 452
SubString function 452
%Substring meta-SQL function 550
Substringb function 453
suggestions, submitting xciii
SwitchUser function 454
synchronization
advanced considerations 1649
event considerations 1635
synchronization server
events 1633
SyncRequestXmlDoc function 455
SyncServer class
methods 1643
properties 1644
scope of 1642
%SyncServer variable 579
System variables 563
%AllowNotification 563
%AllowRecipientLookup 563
%AsOfDate 563
%AuthenticationToken 564
%BPName 564
%ClientDate 564
%ClientTimeZone 564
%CompIntfcName 565
%Component 564
%ContentID 565
%ContentType 565
%Copyright 566
%Currency 566
%Date 566
%DateTime 567
%DbName 567
%DbServerName 567
%DbType 567
%EmailAddress 568
%EmployeeId 568
%ExternalAuthInfo 568
%HPTabName 568
%Import 568
%IsMultiLanguageEnabled 569
%Language 569
%Language_Base 569
%LocalNode 569
%Market 570
%MaxInterlinkSize 570
%MaxMessageSize 570

%Menu 571
%Mode 571
%NavigatorHomePermissionList 571
%Node 571
%OperatorClass 572
%OperatorId 572
%OperatorRowLevelSecurityClass 572
%Page 573
%Panel 573
%PanelGroup 573
%PasswordExpired 574
%PerfTime 574
%PermissionLists 575
%Portal 575
%PrimaryPermissionList 575
%ProcessProfilePermissionList 576
%PSAuthResult 576
%Request 576
%Response 576
%ResultDocument 576
%Roles 577
%RowSecurityPermissionList 577
%RunningInPortal 577
%ServerTimeZone 577
%Session 577
%SignonUserId 578
%SignonUserPswd 578
%SMTPBlackberryReplyTo 578
%SMTPGuaranteed 578
%SMTPSender 579
%SQLRows 579
%SyncServer 579
%Time 580
%TransformData 580
%UserDescription 581
%UserId 581
%WLInstanceID 581
%WLName 581

T

TabDefinition
class properties 1187
collection methods 1192
collection property 1195
%Table meta-SQL function 551
Tan function 456
terms 1949
%Test meta-SQL function 552
%TextIn meta-SQL function 553
%ThisMobileObject system variable 580
Throw language construct 457
Time and date functions
AddToDate 39
AddToDateTime 40
AddToTime 40
ConvertDatetimeToBase 85
ConvertTimeToBase 87
Date 117
Date3 118
%DateAdd 521
%DateDiff 521
%DateNull 522
%DateOut 523
DatePart 118
%DatePart 523
DateTime6 119
%DateTimeDiff 523
%DateTimeIn 524
%DateTimeNull 524
%DateTimeOut 525
DateTimeToHTTP 120
DateTimeToLocalizedString 121
DateTimeToTimeZone 124
DateTimeValue 125
DateValue 126
Day 127
Days 128
Days360 128
Days365 129
%DTTM 526
FormatDateTime 192
GetCalendarDate 237
Hour 291
IsDate 301
IsDateTime 302
IsDaylightSavings 302
IsTime 311
Minute 326
Month 327
Second 393
Time 458
Time3 458
%TimeAdd 553
%TimeIn 554
%TimeNull 554
%TimeOut 555
TimePart 459
%TimePart 555
TimeToTimeZone 460
TimeValue 461

TimeZoneOffset 461
Weekday 501
Year 509
Time function 458
%Time variable 580
Time3 function 458
%TimeAdd meta-SQL function 553
%TimeIn meta-SQL function 554
%TimeNull meta-SQL function 554
%TimeOut meta-SQL function 555
TimePart function 459
%TimePart meta-SQL function 555
TimeToTimeZone function 460
TimeValue function 461
TimeZoneOffset function 461
%ToolsRel meta-HTML 590
TotalRowCount function 462
Trace functions
SetTracePC 432
SetTraceSQL 435
trace settings object
about 1562
properties 1580
Transfer function 463
Transfer functions
AddKeyListItem 36
ClearKeyList 67
DoModalComponent 150
EndModalComponent 162
IsModalComponent 308
SetNextPage 422
Transfer 463
TransferNode 468
TransferPage 475
TransferPortal 471
TransferMobilePage function 467
TransferNode function 468
TransferPage function 475
TransferPortal function 471
Transform function 476
%TransformData variable 580
TransformEx function 478
tree
class methods 1703
class properties 1723
collection method 1737
collection properties 1738
tree classes
branch collection methods 1666
branch collection properties 1667
branch collections 1666
collections in 1656
declaring 1659
error handling 1658
implementing 1660
insertion verification 1659
leaf 1668
leaf methods 1668
leaf properties 1674
level 1680
level collection 1678
level collection methods 1678
level methods 1680
level properties 1681
node class 1683
node methods 1683
node properties 1695
quick reference 1910
relationship between 1654
scope of 1659
session class methods 1663
tree 1702
tree collection 1736
tree collection method 1737
tree collection properties 1738
tree methods 1703
tree properties 1723
tree structure 1739
tree structure collection 1753
tree structure collection method 1754
tree structure collection properties 1755
tree structure methods 1739
tree structure properties 1742
TreeDetailInNode function 480
using tree viewer 1939
tree structure
class methods 1739
class properties 1742
collection method 1754
collection properties 1755
Tree Viewer
invoking 1942
multi-node example 1946
no multi-node example 1945
overview 1939
TreeDetailInNode function 480
TriggerBusinessEvent function 481
%TrimSubstr meta-SQL function 556
Truncate function 481

%Truncate meta-SQL function 556
%Truncatetable meta-SQL function 557
Try language construct 482
try sections 837
type checking functions
IsAlpha 300
IsAlphaNumeric 300
IsDate 301
IsDateTime 302
IsDigits 304
IsNumber 309
IsTime 311
Max 319
Min 326
NumberToString 337
typographical conventions xci

U

Unencode function 483
UnGray function 484
UnHide function 486
UnhideRow function 487
UnhideScroll function 488
UniformSeriesPV function 489
%UpdatePairs meta-SQL function 558
UpDateSysVersion function 490
UpdateValue function 490
UpdateXmlDoc function 492
Upper function 492
%Upper meta-SQL function 559
%URL meta-HTML 590
URL vs. URI 932
User Security functions
AllowEmplIdChg 43
Decrypt 133
EncodeURL 159
ExecuteRolePeopleCode 173
ExecuteRoleQuery 174
ExecuteRoleWorkflowQuery 175
Hash 284
IsMenuItemAuthorized 305
IsUserInPermissionList 312
IsUserInRole 312
RevalidatePassword 379
SetAuthenticationResult 399
SetPasswordExpired 424
SwitchUser 454
%UserDescription variable 581
UserHomepage
class method 1227
class properties 1228
%UserId meta-HTML 590
%UserId variable 581
UserTab
class properties 1229
collection methods 1230
collection property 1233
using
application classes 602
content references 1106
exceptions 836
mobile PeopleCode 1013

V

validation functions
Error 165
RevalidatePassword 379
SetCursorPos 404
SetReEdit 428
Warning 499
Value function 493
variables
declaring 74, 281, 315
ViewAttachment function 494
ViewContentURL function 498
viewing trees 1939
ViewURL function 498
visual cues xcii

W

Warning 499
warnings xciii
web libraries
creating 933
GetMethodNames function 254
security 934
Weekday function 501
While 502
WinEscape function 502
WinExec function 503
WinMessage function 505
%WLInstanceID variable 581
%WLName variable 581
Workflow
notification classes 1061
Workflow functions
GenerateActGuideContentUrl 198
GenerateActGuidePortalUrl 199
GenerateActGuideRelativeUrl 200

GenerateWorklistPortalURL 231
GenerateWorklistRelativeURL 232
GetWLFieldValue 281
MarkWLItemWorked 318
TriggerBusinessEvent 481
Workflow system variables
%AllowNotification 563
%AllowRecipientLookup 563
%BPName 564
%SMTPBlackberryReplyTo 578
%SMTPGuaranteed 578
%SMTPSender 579
Worklist system variables
%WLInstanceID 581
%WLName 581
WorklistEntry
class methods 1077
class properties 1080
creating 1084
updating 1086

X

XmlDoc class
considerations 1759
CreateSOAPDoc function 109
creating 1758
data type 1761
error handling 1760
methods 1762
property 1775
scope of 1761
SOAPDoc considerations 1590, 1760
using 1758
when to use 1758
XmlDoc functions
CreateSOAPDoc 109
CreateXmlDoc 111
GetMessageXmlDoc 255
GetNRXmlDoc 261
PublishXmlDoc 352
ReSubmitPubHeaderXmlDoc 374
ReSubmitPubXmlDoc 375
ReSubmitSubXmlDoc 376
ReValidateNRXmlDoc 379
SyncRequestXmlDoc 455
Transform 476
TransformEx 478
UpdateXmlDoc 492
XmlNode class
data type 1761
methods 1776
properties 1808
scope of 1761

Y

Year function 509