PeopleTools 8.4: PeopleCode Reference

PeopleSoft®

PeopleTools 8.4: PeopleCode Reference

SKU Tr84PCR-B 0302

PeopleBooks Contributors: Teams from PeopleSoft Product Documentation and Development.

Copyright © 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, the PeopleSoft logo, PeopleTools, PS/nVision, PeopleCode, PeopleBooks, *PeopleTalk,* and Vantive are registered trademarks, and "People power the internet." and Pure Internet Architecture 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.

Contents

PeopleCode Reference Preface

About This PeopleBook lxxxi
Before You Begin lxxxi
PeopleSoft Application Fundamentals lxxxii
Related Documentation lxxxii
    Hard-copy Documentation lxxxiii
PeopleBooks Standard Field Definitions lxxxiii
Typographical Conventions and Visual Cues lxxxv
Page and Panel Introductory Table lxxxvi
Comments and Suggestions lxxxvi

Chapter 1

PeopleCode Built-in Functions

PeopleCode Typographical Conventions 1-1
Functions by Category 1-2
    APIs 1-2
    Application Classes 1-2
    Application Engine 1-2
    Arrays 1-2
    Attachment 1-2
    Bulk 1-2
    Business Interlink 1-3
    Chart 1-3
    ChartField 1-3
    Component Buffer 1-3
    Component Interface 1-4
    Conversion 1-4
    Currency and Financial 1-4
    Current Date and Time 1-5
    Custom Display Formats 1-5
    Database and Platform 1-5
    Data Buffer Access 1-5
    Date and Time 1-5
    Defaults, Setting 1-6

Double-Byte Characters 1-6
Effective Date and Effective Sequence 1-7
Email 1-7
Environment 1-7
Exceptions 1-7
Executable Files, Running 1-7
Files 1-7
Financial 1-8
Grids 1-8
Images 1-8
Import Manager 1-8
Integration Broker 1-8
Internet 1-9
Java 1-9
Language Preference and Locale 1-9
Language Constructs 1-10
Logical (Tests for Blank Values) 1-10
Mail 1-10
Math 1-10
Menu Appearance 1-11
Message Catalog and Message Display 1-11
Messages 1-11
Modal Transfers 1-11
Object-Based 1-12
Object 1-13
Page 1-13
Page Control Appearance 1-13
Personalizations 1-13
Process Scheduler 1-13
Remote Call 1-13
Saving and Canceling 1-14
Scroll Select 1-14
Search Dialog 1-14
Secondary Pages 1-14
SQL 1-14
SQL Date and Time 1-15
SQL Shortcuts 1-15
String 1-15
Subrecords 1-16
TimeZone 1-16

Trace Control 1-16
Transfers 1-17
Type Checking 1-17
User Information 1-17
User Security 1-17
Validation 1-18
Workflow 1-18
XML 1-18
PeopleCode Built-in Functions and Language Constructs A-C 1-19
Abs 1-19
AccruableDays 1-19
AccrualFactor 1-20
Acos 1-21
ActiveRowCount 1-22
AddAttachment 1-23
AddDropDownItem 1-28
AddEmailAddress 1-30
AddKeyListItem 1-30
AddSystemPauseTimes 1-31
AddToDate 1-33
AddToDateTime 1-34
AddToTime 1-35
All 1-36
AllOrNone 1-37
AllowEmplIdChg 1-38
Amortize 1-39
Asin 1-40
Atan 1-41
BlackScholesCall 1-41
BlackScholesPut 1-42
BootstrapYTMs 1-43
Break 1-44
BulkDeleteField 1-44
BulkInsertField 1-47
BulkModifyPageFieldOrder 1-50
CallAppEngine 1-52
CancelPubHeaderXmlDoc 1-55
CancelPubXmlDoc 1-56
CancelSubXmlDoc 1-57
ChangeEmailAddress 1-58

Char 1-59
CharType 1-59
ChDir 1-61
ChDrive 1-61
CheckMenuItem 1-62
Clean 1-62
ClearDropDownList 1-63
ClearKeyList 1-63
ClearSearchDefault 1-64
ClearSearchEdit 1-65
Code 1-65
Codeb 1-66
CollectGarbage 1-66
CommitWork 1-67
CompareLikeFields 1-70
Component 1-71
ComponentChanged 1-72
ContainsCharType 1-73
ContainsOnlyCharType 1-75
ConvertChar 1-77
ConvertCurrency 1-81
ConvertDatetimeToBase 1-83
ConvertRate 1-84
ConvertTimeToBase 1-86
CopyAttachments 1-87
CopyFields 1-88
CopyRow 1-90
Cos 1-90
Cot 1-91
CreateArray 1-92
CreateArrayAny 1-93
CreateArrayRept 1-94
CreateDirectory 1-95
CreateException 1-97
CreateJavaArray 1-98
CreateJavaObject 1-99
CreateMessage 1-100
CreateObject 1-101
CreateObjectArray 1-104
CreateProcessRequest 1-106

CreateRecord 1-106
CreateRowset 1-108
CreateSOAPDoc 1-110
CreateSQL 1-111
CreateXmlDoc 1-113
CubicSpline 1-114
CurrEffDt 1-116
CurrEffRowNum 1-116
CurrEffSeq 1-117
CurrentLevelNumber 1-118
CurrentRowNumber 1-118
PeopleCode Built-in Functions and Language Constructs D-G 1-120
Date 1-120
Date3 1-120
DatePart 1-121
DateTime6 1-121
DateTimeToHTTP 1-122
DateTimeToLocalizedString 1-124
DateTimeToTimeZone 1-126
DateTimeValue 1-127
DateValue 1-128
Day 1-130
Days 1-130
Days360 1-131
Days365 1-131
DBCSTrim 1-132
DBPatternMatch 1-132
Declare Function 1-133
Decrypt 1-137
Degrees 1-137
DeleteAttachment 1-138
DeleteEmailAddress 1-141
DeleteImage 1-142
DeleteRecord 1-143
DeleteRow 1-144
DeleteSQL 1-145
DeleteSystemPauseTimes 1-147
DisableMenuItem 1-149
DiscardRow 1-149
DoCancel 1-151

DoModal 1-151
DoModalComponent 1-154
DoModalPanelGroup 1-157
DoSave 1-157
DoSaveNow 1-158
EnableMenuItem 1-160
EncodeURL 1-161
EncodeURLForQueryString 1-161
Encrypt 1-162
EncryptNodePswd 1-163
EndMessage 1-164
EndModal 1-165
EndModalComponent 1-166
Error 1-167
EscapeHTML 1-169
EscapeJavascriptString 1-170
EscapeWML 1-171
Evaluate 1-171
Exact 1-172
Exec 1-173
ExecuteRolePeopleCode 1-176
ExecuteRoleQuery 1-177
ExecuteRoleWorkflowQuery 1-178
Exit 1-178
Exp 1-179
ExpandBindVar 1-180
ExpandEnvVar 1-181
ExpandSqlBinds 1-181
Fact 1-182
FetchSQL 1-183
FetchValue 1-184
FieldChanged 1-186
FileExists 1-187
Find 1-189
Findb 1-190
FindCodeSetValues 1-191
FindFiles 1-193
FlushBulkInserts 1-195
For 1-196
FormatDateTime 1-197

Function 1-198
GenerateActGuideContentUrl 1-200
GenerateActGuidePortalUrl 1-201
GenerateActGuideRelativeUrl 1-203
GenerateComponentContentRelURL 1-204
GenerateComponentContentURL 1-206
GenerateComponentPortalRelURL 1-209
GenerateComponentPortalURL 1-211
GenerateComponentRelativeURL 1-214
GenerateExternalPortalURL 1-216
GenerateExternalRelativeURL 1-217
GenerateHomepagePortalURL 1-218
GenerateHomepageRelativeURL 1-219
GenerateQueryContentURL 1-220
GenerateQueryPortalURL 1-222
GenerateQueryRelativeURL 1-224
GenerateScriptContentRelURL 1-226
GenerateScriptContentURL 1-227
GenerateScriptPortalRelURL 1-229
GenerateScriptPortalURL 1-231
GenerateScriptRelativeURL 1-232
GenerateTree 1-234
GenerateWorklistPortalURL 1-235
GenerateWorklistRelativeURL 1-236
GetAESection 1-238
GetAttachment 1-239
GetBiDoc 1-242
GetCalendarDate 1-243
GetChart 1-245
GetChartURL 1-246
GetCwd 1-247
GetEnv 1-247
GetField 1-248
GetFile 1-249
GetGrid 1-253
GetHTMLText 1-255
GetImageExtents 1-256
GetInterlink 1-257
GetJavaClass 1-259
GetLevel0 1-260

GetMethodNames 1-261
GetMessage 1-262
GetMessageInstance 1-263
GetMessageXmlDoc 1-263
GetNextNumber 1-265
GetNextNumberWithGaps 1-267
GetNextProcessInstance 1-268
GetNRXmlDoc 1-269
GetPage 1-270
GetPubContractInstance 1-271
GetPubHeaderXmlDoc 1-272
GetPubXmlDoc 1-273
GetRecord 1-273
GetRelField 1-275
GetRow 1-276
GetRowset 1-277
GetSelectedTreeNode 1-278
GetSession 1-278
GetSetId 1-279
GetSQL 1-280
GetStoredFormat 1-283
GetSubContractInstance 1-284
GetSubXmlDoc 1-286
GetSyncLogData 1-287
GetTreeNodeParent 1-288
GetTreeNodeRecordName 1-288
GetTreeNodeValue 1-288
GetURL 1-289
GetUserOption 1-290
GetWLFieldValue 1-290
Global 1-291
Gray 1-292
GrayMenuItem 1-293
PeopleCode Built-in Functions and Language Constructs H-R 1-295
Hash 1-295
HermiteCubic 1-296
Hide 1-296
HideMenuItem 1-298
HideRow 1-298
HideScroll 1-300

HistVolatility 1-301
Hour 1-302
Idiv 1-303
If 1-303
InsertImage 1-304
InsertRow 1-306
Int 1-307
Integer 1-308
IsAlpha 1-309
IsAlphaNumeric 1-309
IsDate 1-310
IsDateTime 1-311
IsDaylightSavings 1-312
IsDigits 1-313
IsHidden 1-313
IsMenuItemAuthorized 1-314
IsMessageActive 1-316
IsModal 1-317
IsModalComponent 1-318
IsModalPanelGroup 1-318
IsNumber 1-319
IsOperatorInClass 1-319
IsSearchDialog 1-320
IsTime 1-320
IsUserInPermissionList 1-321
IsUserInRole 1-322
Left 1-322
Len 1-323
Lenb 1-323
LinearInterp 1-324
Ln 1-324
Local 1-325
Log10 1-326
Lower 1-326
LTrim 1-327
MarkPrimaryEmailAddress 1-328
MarkWLItemWorked 1-328
Max 1-329
MessageBox 1-330
Min 1-336

Minute 1-336
Mod 1-337
Month 1-337
MsgGet 1-338
MsgGetExplainText 1-339
MsgGetText 1-340
NextEffDt 1-341
NextRelEffDt 1-342
NodeDelete 1-342
NodeRename 1-344
NodeSaveAs 1-344
None 1-345
NumberToString 1-346
ObjectDoMethod 1-349
ObjectDoMethodArray 1-350
ObjectGetProperty 1-351
ObjectSetProperty 1-352
OnlyOne 1-353
OnlyOneOrNone 1-354
PanelGroupChanged 1-355
PingNode 1-356
PriorEffDt 1-357
PriorRelEffDt 1-358
PriorValue 1-358
Product 1-359
Prompt 1-359
Proper 1-361
PublishXmlDoc 1-362
PutAttachment 1-363
Radians 1-367
Rand 1-367
RecordChanged 1-368
RecordDeleted 1-370
RecordNew 1-372
RefreshTree 1-374
RemoteCall 1-374
RemoveDirectory 1-377
RenameDBField 1-379
RenamePage 1-380
RenameRecord 1-382

Repeat 1-383
Replace 1-383
Rept 1-384
ReSubmitPubHeaderXmlDoc 1-385
ReSubmitPubXmlDoc 1-386
ReSubmitSubXmlDoc 1-387
Return 1-388
ReturnToServer 1-388
ReValidateNRXmlDoc 1-390
RevalidatePassword 1-391
Right 1-392
Round 1-393
RoundCurrency 1-394
RowFlush 1-394
RowScrollSelect 1-395
RowScrollSelectNew 1-397
RTrim 1-399
PeopleCode Built-in Functions and Language Constructs S-Z 1-401
ScheduleProcess 1-401
ScrollFlush 1-401
ScrollSelect 1-402
ScrollSelectNew 1-406
Second 1-407
SendMail 1-408
SetAuthenticationResult 1-412
SetDBFieldAuxFlag 1-413
SetDBFieldCharDefn 1-414
SetDBFieldFormat 1-416
SetDBFieldFormatLength 1-418
SetDBFieldLabel 1-420
SetDBFieldLength 1-421
SetDBFieldNotUsed 1-423
SetChannelStatus 1-424
SetControlValue 1-425
SetCursorPos 1-427
SetDefault 1-429
SetDefaultAll 1-430
SetDefaultNext 1-430
SetDefaultNextRel 1-431
SetDefaultPrior 1-431

SetDefaultPriorRel 1-432
SetDisplayFormat 1-432
SetLabel 1-433
SetLanguage 1-434
SetNextPanel 1-436
SetNextPage 1-436
SetPageFieldPageFieldName 1-437
SetPasswordExpired 1-438
SetRecFieldEditTable 1-439
SetRecFieldKey 1-441
SetReEdit 1-442
SetSearchDefault 1-443
SetSearchDialogBehavior 1-444
SetSearchEdit 1-445
SetTempTableInstance 1-446
SetTracePC 1-447
SetTraceSQL 1-450
SetupScheduleDefnItem 1-452
SetUserOption 1-453
Sign 1-453
Sin 1-454
SinglePaymentPV 1-454
SortScroll 1-455
Split 1-456
SQLExec 1-457
Sqrt 1-463
StartWork 1-463
StopFetching 1-464
StoreSQL 1-465
String 1-467
Substitute 1-468
Substring 1-469
Substringb 1-469
SwitchUser 1-470
SyncRequestXmlDoc 1-472
Tan 1-473
Throw 1-474
Time 1-475
Time3 1-475
TimePart 1-476

TimeToTimeZone 1-477
TimeValue 1-478
TimeZoneOffset 1-478
TotalRowCount 1-479
Transfer 1-480
TransferNode 1-484
TransferPortal 1-487
TransferPanel 1-489
TransferPage 1-489
Transform 1-491
TreeDetailInNode 1-492
TriggerBusinessEvent 1-493
Truncate 1-494
Try 1-494
UnCheckMenuItem 1-495
Unencode 1-496
Ungray 1-496
Unhide 1-498
UnhideRow 1-500
UnhideScroll 1-501
UniformSeriesPV 1-502
UpdateSysVersion 1-503
UpdateValue 1-503
UpdateXmlDoc 1-504
Upper 1-505
Value 1-506
ViewAttachment 1-506
ViewContentURL 1-511
ViewURL 1-512
Warning 1-513
Weekday 1-515
While 1-515
WinEscape 1-516
WinExec 1-517
WinMessage 1-519
Year 1-523

Chapter 2

Meta-SQL

Date Considerations 2-1

Use of Date, Datetime, or Time Wrappers with an Application Engine Program 2-2
Date and Time Out Wrappers for Static and Dynamic Views 2-2
Using {DateTimein-prefix} in SQR 2-2
Meta-SQL Placement Considerations 2-3
Meta-SQL Reference 2-6
%Abs 2-6
%BINARYSORT 2-6
%Concat 2-7
%CurrentDateIn 2-8
%CurrentDateOut 2-8
%CurrentDateTimeIn 2-8
%CurrentDateTimeOut 2-8
%CurrentTimeIn 2-8
%CurrentTimeOut 2-8
%DateAdd 2-9
%DateDiff 2-9
%DateIn 2-9
%DateNull 2-10
%DateOut 2-10
%DatePart 2-11
%DateTimeDiff 2-11
%DateTimeIn 2-12
%DateTimeNull 2-12
%DateTimeOut 2-13
%DecDiv 2-13
%DecMult 2-14
%DTTM 2-14
%EffDtCheck 2-15
%FirstRows 2-16
%InsertSelect 2-18
%InsertValues 2-20
%Join 2-21
%KeyEqual 2-23
%KeyEqualNoEffDt 2-24
%Like 2-25
%LikeExact 2-27
%List 2-29
%ListBind 2-32
%ListEqual 2-33
%NoUppercase 2-34

%OldKeyEqual 2-35
%OPRCLAUSE 2-36
%Round 2-36
%SQL 2-37
%Substring 2-39
%SUBREC 2-39
%Table 2-40
%Test 2-41
%TextIn 2-41
%TimeAdd 2-42
%TimeIn 2-43
%TimeNull 2-43
%TimeOut 2-44
%TimePart 2-44
%TrimSubstr 2-44
%Truncate 2-45
%TruncateTable 2-45
%UpdatePairs 2-46
%Upper 2-47
Meta-SQL Shortcuts 2-48
%Delete 2-49
%Insert 2-49
%SelectAll 2-49
%SelectDistinct 2-50
%SelectByKey 2-51
%SelectByKeyEffDt 2-51
%Update 2-51

Chapter 3

System Variables

Understanding System Variables 3-1
%AllowNotification 3-1
%AllowRecipientLookup 3-1
%AsOfDate 3-2
%AuthenticationToken 3-2
%BPName 3-2
%ClientDate 3-2
%ClientTimeZone 3-2
%Component 3-3
%CompIntfcName 3-3

%ContentID 3-3
%ContentType 3-3
%Copyright 3-4
%Currency 3-4
%Date 3-4
%DateTime 3-4
%DbName 3-4
%DbServerName 3-5
%DbType 3-5
%EmailAddress 3-5
%EmployeeId 3-5
%ExternalAuthInfo 3-5
%FilePath 3-5
%HPTabName 3-6
%Import 3-6
%IsMultiLanguageEnabled 3-6
%Language 3-6
%Language_Base 3-6
%LocalNode 3-6
%Market 3-7
%MaxInterlinkSize 3-8
%MaxMessageSize 3-8
%Menu 3-8
%Mode 3-8
%NavigatorHomePermissionList 3-8
%Node 3-9
%OperatorClass 3-9
%OperatorId 3-9
%OperatorRowLevelSecurityClass 3-9
%OutDestFormat 3-9
%OutDestType 3-10
%Page 3-10
%Panel 3-10
%PanelGroup 3-10
%PasswordExpired 3-11
%PermissionLists 3-11
%Portal 3-11
%PrimaryPermissionList 3-11
%ProcessProfilePermissionList 3-11
%PSAuthResult 3-12

%Request 3-12
%Response 3-12
%ResultDocument 3-12
%Roles 3-13
%RowSecurityPermissionList 3-13
%RunningInPortal 3-13
%ServerTimeZone 3-13
%Session 3-13
%SignonUserId 3-13
%SignOnUserPswd 3-14
%SMTPBlackberryReplyTo 3-14
%SMTPGuaranteed 3-14
%SMTPSender 3-15
%SQLRows 3-15
%SyncServer 3-15
%Time 3-15
TransformData 3-16
%UserDescription 3-16
%UserId 3-16
%WLInstanceID 3-16
%WLName 3-16

Chapter 4

Meta-HTML

Understanding Meta-HTML 4-1
Variables 4-2
Functions 4-2
Comments 4-2
Alphabetical List of Meta-HTML Elements 4-3
%Appserver 4-3
%AppsRel 4-3
%Browser 4-3
%BrowserPlatform 4-3
%BrowserVersion 4-4
%Cols 4-4
%Component 4-4
%Copyright 4-4
%DBName 4-4
%DBType 4-5
%Encode 4-5

%Formname 4-5
%Image 4-5
%JavaScript 4-6
%LabelTag 4-6
%Menu 4-6
%Message 4-7
%Page 4-8
%ServicePack 4-8
%SubmitScriptName 4-8
%ToolsRel 4-8
%URL 4-8
%UserId 4-9

Chapter 5

AESection Class

Understanding AESection Class 5-1
How an AESection is Accessed 5-2
AESection Example 5-4
Declaring an AESection Object 5-5
Scope of an AESection Object 5-5
AESection Class Built-in Function 5-5
AESection Class Methods 5-5
AddStep 5-5
Close 5-6
Open 5-7
Save 5-8
SetSQL 5-9
SetTemplate 5-10
AESection Class Property 5-11
IsOpen 5-11

Chapter 6

Application Classes

Understanding Application Classes 6-1
When Would You Use Application Classes? 6-2
Application Classes General Structure 6-2
Class Name 6-3
Class Extension 6-4
Declaration of Public External Interface 6-5
Declaration of Private Instance Variables and Methods 6-5

Definition of Methods 6-7
Constructors 6-8
External Function Declarations 6-10
Import Declarations 6-11
Self-Reference 6-12
Property Overrides 6-14
Superclass Reference 6-15
Downcasting 6-16
Exception Handling 6-18
Declaration of Application Classes 6-18
Scope of an Application Class 6-18
Application Classes Built-in Functions and Language Constructs 6-18
Class 6-18
Get 6-20
Import 6-21
Method 6-22
Set 6-23

Chapter 7

Array Class

Understanding Arrays 7-1
Creating Arrays 7-1
Populating an Array 7-3
Removing Items From an Array 7-4
Creating Empty Arrays 7-5
Creating and Populating Multi-Dimensional Arrays 7-5
Using Flattening and Promotion 7-8
Declaring Array Objects 7-9
Understanding the Scope of an Array Object 7-9
Array Class Built-in Functions 7-9
Array Class Methods 7-10
Clone 7-10
Find 7-11
Get 7-12
Join 7-12
Next 7-13
Pop 7-15
Push 7-16
Replace 7-17
Reverse 7-19
Set 7-20

Shift 7-21
Sort 7-21
Subarray 7-23
Substitute 7-24
Unshift 7-25
Array Class Properties 7-26
Dimension 7-26
Len 7-26

Chapter 8

Business Interlink Class

Understanding Business Interlink Class 8-1
Using the Interlink Object 8-2
Deciding Which Methods to Use 8-2
Executing the Business Interlink Object 8-3
Supporting Batch Input and Output 8-3
Supporting Rowsets 8-3
Using the Flat Table Methods 8-3
Supporting Dynamic Output 8-3
Using Hierarchical Data (BIDocs) 8-4
Using the Incoming Business Interlink Methods and Properties 8-8
Understanding the State of an Interlink Object 8-9
Using Business Interlink with Application Engine 8-9
Using the Data Type of a Business Interlink Object 8-9
Understanding the Scope of a Business Interlink Object 8-10
Business Interlink Class Built-in Function 8-10
Business Interlink Class Methods 8-10
AddDoc 8-10
AddInputRow 8-12
AddNextDoc 8-13
AddValue 8-16
BulkExecute 8-18
Clear 8-23
Execute 8-23
FetchIntoRecord 8-25
FetchIntoRowset 8-27
FetchNextRow 8-30
GetCount 8-31
GetDoc 8-33
GetFieldCount 8-35

GetFieldType 8-36
GetFieldValue 8-38
GetInputDocs 8-39
GetNextDoc 8-40
GetOutputDocs 8-42
GetPreviousDoc 8-43
GetStatus 8-45
GetValue 8-46
InputRowset 8-48
MoveFirst 8-50
MoveNext 8-51
MoveToDoc 8-52
ResetCursor 8-54
Business Interlink BIDocs Methods 8-55
AddAttribute 8-55
AddComment 8-56
AddProcessInstruction 8-57
AddText 8-58
CreateElement 8-60
GenXMLString 8-61
GetAttributeName 8-61
GetAttributeValue 8-62
GetNode 8-63
ParseXMLString 8-65
Business Interlink BIDocs Properties 8-65
AttributeCount 8-66
ChildNodeCount 8-66
NodeName 8-67
NodeType 8-68
NodeValue 8-69
Business Interlink Class Property 8-70
StopAtError 8-70
Configuration Parameters 8-70
Interlink Plug-in Configuration Parameters 8-70
URL Configuration Parameter 8-71
BIDocValidate Configuration Parameter 8-72

Chapter 9

Chart Class

Understanding Chart Class 9-1

Chart Terms 9-3
Chart Types 9-4
2D Bar 9-5
2D Histogram 9-5
2D Horizontal Bar 9-6
2D Horizontal Stacked Bar 9-6
2D Horizontal Percent Bar 9-7
2D Line 9-7
2D Percent Bar 9-8
2D Pie 9-8
2D Stacked Bar 9-9
3D Bar 9-9
3D Percent Bar 9-10
3D Pie 9-10
3D Stacked Bar 9-11
Style Sheets and Charts 9-11
Considerations Using Style Sheets 9-12
Default Style Sheet Classes 9-12
Component Processor Considerations 9-12
Translation Considerations 9-13
iScripts and Charts 9-13
Error Handling 9-14
Data Type of a Chart Object 9-14
Scope of a Chart Object 9-14
Chart Class Built-in Functions 9-14
Chart Class Reference 9-14
Chart Data Methods 9-15
Refresh 9-15
SetData 9-16
SetDataSeries 9-17
SetDataHints 9-18
SetDataURLs 9-21
SetDataXAxis 9-21
SetDataYAxis 9-22
Chart Data Properties 9-23
DataStartRow 9-23
DataWidth 9-24
ImageMap 9-24
Chart Appearance Properties 9-24
GridLines 9-25

Height ... 9-25
IsDrillable ... 9-25
IsPlainImage ... 9-26
LineType ... 9-26
RotationAngle ... 9-27
Style ... 9-27
StyleSheet ... 9-27
Type ... 9-28
Width ... 9-28
Chart Colors ... 9-29
Chart Color Methods ... 9-29
SetColorArray ... 9-30
SetDataColor ... 9-31
SetOLDataYAxis ... 9-31
Chart Axes Methods ... 9-32
SetXAxisLabels ... 9-32
SetYAxisLabels ... 9-32
Chart Axes Properties ... 9-33
XAxisLabelOrient ... 9-33
XAxisStyle ... 9-33
YAxisLabelOrient ... 9-34
YAxisMax ... 9-34
YAxisMin ... 9-34
YAxisStyle ... 9-35
YAxisTicks ... 9-35
Chart Title Properties ... 9-35
MainTitle ... 9-35
MainTitleOrient ... 9-35
MainTitleStyle ... 9-36
XAxisTitle ... 9-36
XAxisTitleOrient ... 9-36
XAxisTitleStyle ... 9-36
YAxisTitle ... 9-36
YAxisTitleOrient ... 9-37
YAxisTitleStyle ... 9-37
Chart Legend Methods ... 9-37
SetLegend ... 9-37
Chart Legend Properties ... 9-38
HasLegend ... 9-38
LegendMaxEntries ... 9-39

LegendPosition 9-39
LegendStyle 9-39
Chart Overlay Methods 9-40
SetOLData 9-40
SetOLDataSeries 9-40
SetOLDataXAxis 9-41
SetOLDataYAxis 9-41
Chart Overlay Properties 9-42
OLLineType 9-42
OLType 9-42
Chart Examples 9-43
Creating a Chart 9-43
Creating a 'Gantt' Chart 9-45
Creating a Chart Using an iScript 9-48

Chapter 10

Component Interface Classes

Understanding Component Interface Class 10-1
Life Cycle of a Component Interface 10-2
Setting Component Interface Keys 10-4
Standard and User-Defined Component Interface Methods 10-5
Accessing Component Interface Standard Properties 10-6
Accessing User-Defined Component Interface Properties 10-7
Declaring a Component Interface Object 10-7
Scope of a Component Interface Object 10-8
Implementing a Component Interface 10-8
Create a New Instance of Data Example 10-9
Getting an Existing Instance of Data Example 10-12
Retrieving a List of Instance of Data Example 10-15
Component Interface Methods and Timeouts 10-17
Traversing a Component Interface and Using Data Collections 10-18
Inserting Effective-Dated Data Example 10-20
Inserting Effective-Dated Data Example Using Visual Basic 10-22
Inserting or Deleting a Row of Data Example 10-24
Working With Effective-Dated Data 10-28
How a Developer Uses GetEffectiveItem 10-29
Reusing Existing Code 10-29
Differences in Search Dialog Processing 10-30
Differences in PeopleCode Event and Function Behavior 10-30
Using %Menu Conditions 10-30

Session Object Methods 10-31
FindCompIntfcs 10-31
GetCompIntfc 10-31
Component Interface Collection Methods 10-32
First 10-32
Item 10-33
Next 10-33
Component Interface Collection Property 10-33
Count 10-34
Component Interface Class Methods 10-34
Cancel 10-34
CopyRowset 10-34
CopyRowsetDelta 10-37
CopySetupRowset 10-39
CopySetupRowsetDelta 10-42
Create 10-45
Find 10-45
Get 10-46
GetPropertyByName 10-47
Save 10-48
SetPropertyByName 10-49
Component Interface Class Properties 10-50
ComponentName 10-50
CreateKeyInfoCollection 10-50
EditHistoryItems 10-51
FindKeyInfoCollection 10-51
GetHistoryItems 10-52
GetKeyInfoCollection 10-52
InteractiveMode 10-53
PropertyInfoCollection 10-53
StopOnFirstError 10-53
Data Collection Methods 10-53
CurrentItem 10-54
DeleteItem 10-54
GetEffectiveItem 10-56
GetEffectiveItemNum 10-56
InsertItem 10-57
Item 10-58
ItemByKeys 10-59
Data Collection Properties 10-60

Count 10-60
CurrentItemNumber 10-61
Data Item Class Property 10-61
ItemNum 10-61
Accessing the Structure of a Component Interface 10-61
CompIntfPropInfoCollection Collection 10-62
CompIntfPropInfoCollection Collection Methods 10-64
First 10-64
Item 10-64
Next 10-65
CompIntfPropInfoCollection Collection Properties 10-65
Count 10-65
CompIntfPropInfoCollection Object Properties 10-66
Format 10-66
IsCollection 10-66
IsReadOnly 10-67
Key 10-67
LabelLong 10-67
LabelShort 10-67
Length 10-67
Name 10-68
Prompt 10-68
PropertyInfoCollection 10-68
Required 10-68
Type 10-69
Xlat 10-69
YesNo 10-69

Chapter 11

Exception Class

Understanding Exception Class 11-1
How Exceptions Are Thrown 11-2
When to Use Exceptions 11-2
Try-Catch Sections 11-3
Data Type for Exception Class Objects 11-4
Scope of Exception Class Objects 11-4
Exception Class Built-in Function and Language Constructs 11-4
Exception Class Methods 11-4
GetSubstitution 11-4
Output 11-5

SetSubstitution 11-6
ToString 11-6
Exception Class Properties 11-7
Context 11-7
DefaultText 11-7
MessageNumber 11-8
MessageSetNumber 11-8
MessageSeverity 11-8
SubstitutionCount 11-8

Chapter 12

Field Class

Field Class Overview 12-1
Shortcut Considerations 12-1
Data Type for a Field Object 12-2
Scope of a Field Object 12-2
Field Class Built-in Function 12-2
Field Class Methods 12-2
GetAuxFlag 12-3
GetLongLabel 12-3
GetRelated 12-4
GetShortLabel 12-6
SearchClear 12-7
SetCursorPos 12-7
SetDefault 12-9
Field Class Properties 12-9
DataAreaCollapsed 12-10
DisplayFormat 12-10
DisplayOnly 12-12
EditError 12-12
Enabled 12-12
FieldLength 12-13
FormatLength 12-13
FormattedValue 12-13
IsAltKey 12-14
IsAuditFieldAdd 12-14
IsAuditFieldChg 12-14
IsAuditFieldDel 12-15
IsAutoUpdate 12-15
IsChanged 12-15

IsDateRangeEdit 12-15
IsDescKey 12-15
IsDuplKey 12-16
IsEditTable 12-16
IsEditXlat 12-16
IsFromSearchField 12-16
IsInBuf 12-16
IsKey 12-17
IsListItem 12-17
IsRequired 12-17
IsSearchItem 12-17
IsSystem 12-18
IsThroughSearchField 12-18
IsUseDefaultLabel 12-18
IsYesNo 12-18
Label 12-18
LabelImage 12-19
LongTranslateValue 12-20
MessageNumber 12-20
MessageSetNumber 12-21
Name 12-21
OriginalValue 12-21
ParentRecord 12-22
PromptTableName 12-22
SearchDefault 12-22
SearchEdit 12-23
ShortTranslateValue 12-24
ShowRequiredFieldCue 12-24
SqlText 12-25
StoredFormat 12-25
Style 12-26
Type 12-26
Value 12-27
Visible 12-28

Chapter 13

File Class

Data Type of a File Object 13-1
Scope of a File Object 13-2
File Layout 13-2

File Security Considerations 13-3
File Access Interruption Recovery 13-3
Plain Text Files 13-4
Automatic PeopleCode Generation 13-5
File Layout Error Processing 13-6
File Class Built-in Functions 13-7
File Class Methods 13-8
Close 13-8
CreateRowset 13-8
GetPosition 13-9
Open 13-11
ReadLine 13-14
ReadRowset 13-15
SetFileId 13-17
SetFileLayout 13-19
SetPosition 13-20
WriteLine 13-22
WriteRaw 13-23
WriteRecord 13-24
WriteRowset 13-25
WriteString 13-27
File Class Properties 13-28
CurrentRecord 13-28
IgnoreInvalidId 13-28
IsError 13-29
IsNewFileId 13-29
IsOpen 13-30
Name 13-30
RecTerminator 13-31
File Layout Examples 13-31
WriteRecord Example 13-32
ReadRecord Example 13-35
WriteRowset Example 13-37
ReadRowset Example 13-40
File Rowset Considerations 13-41
Application Engine Example 13-41
Multiple File Layouts 13-44
Reading Multiple File Layouts 13-45
Writing Multiple File Layouts 13-48

Chapter 14

Grid Classes

Understanding Grid and GridColumn Classes 14-1
Shortcut Considerations 14-2
The Grid Class in PeopleCode 14-2
Data Type for a Grid or Grid Column Object 14-3
Scope of a Grid or Grid Column Object 14-3
Grid Class Built-In Function 14-3
Grid Class Method 14-4
    GetColumn 14-4
Grid Class Property 14-6
    gridcolumn 14-6
    Label 14-6
GridColumn Class 14-6
GridColumn Class Properties 14-7
    Enabled 14-7
    Label 14-7
    Name 14-8
    Visible 14-8

Chapter 15

Internet Script Classes (iScript)

Understanding Internet Script Classes 15-1
URL vs. URI 15-3
Web Libraries 15-3
iScript Security 15-5
When to Use an iScript 15-5
    Style Sheets and Styles 15-6
    Other Considerations 15-6
Details of an iScript URL 15-7
The Generate Functions 15-7
Error Handling 15-9
Scope of the Internet Script Classes 15-9
Data Types of the iScript Classes 15-9
Internet Script Classes Built-in Functions 15-10
Internet Script Reference 15-11
    Request Class 15-11
    Response 15-11
    Cookies 15-12
Request Class Methods 15-12

GetContentBody 15-12
GetCookieNames 15-13
GetCookieValue 15-13
GetHeader 15-13
GetHeaderNames 15-14
GetHelpURL 15-14
GetParameter 15-14
GetParameterNames 15-15
GetParameterValues 15-15
Request Class Properties 15-15
AuthTokenDomain 15-15
ByPassSignOn 15-16
BrowserPlatform 15-16
BrowserType 15-16
BrowserVersion 15-16
ContentURI 15-16
ExpireMeta 15-17
FullURI 15-17
HTTPMethod 15-18
LogoutURL 15-18
PathInfo 15-18
PortalURI 15-18
Protocol 15-19
QueryString 15-19
RelativeURL 15-19
RemoteAddr 15-19
RemoteHost 15-19
RequestURI 15-20
Scheme 15-20
ServerName 15-20
ServerPort 15-21
Timeout 15-21
Response Class 15-21
Response Class Methods 15-21
Clear 15-22
CreateCookie 15-22
GetCookie 15-22
GetCookieNames 15-22
GetHeader 15-23
GetHeaderNames 15-23

GetImageURL 15-23
GetJavaScriptURL 15-24
GetStyleSheetURL 15-25
RedirectURL 15-25
SetContentType 15-25
SetHeader 15-26
UseSimpleURL 15-26
Write 15-27
WriteLine 15-27
Response Class Property 15-28
Charset 15-28
Cookie Class 15-28
Cookie Class Properties 15-28
Domain 15-28
MaxAge 15-29
Name 15-29
Path 15-29
Secure 15-29
Value 15-30

Chapter 16

Java Class

Understanding Java Class 16-1
Supported Versions of Java 16-2
Naming Standards for Classes and Packages 16-2
System Setup for Java Classes 16-2
From PeopleCode to Java 16-4
State Management Concerns 16-4
CreateJavaObject Example 16-5
CreateJavaArray Example 16-6
GetJavaClass Example 16-6
From Java to PeopleCode 16-7
SysVar Java Class 16-7
SysCon Java Class 16-7
Func Java Class 16-7
Name Java Class 16-7
Using PeopleCode Objects 16-8
PeopleCode and Java Data Types Mapping 16-9
Considerations When Using the PeopleCode Java Functions 16-10
Error Handling and the PeopleCode Java Functions 16-11

The Java Debugging Environment 16-11
Data Type of a Java Object 16-12
Scope of a Java Object 16-12
PeopleCode Java Built-in Functions 16-13

Chapter 17

Message Class

Understanding Message Class 17-1
Messaging PeopleCode Events 17-2
Data Type of a Message Object 17-2
Scope of a Message Object 17-3
Message Object Population 17-3
Considerations When Populating a Rowset From a Message 17-5
Considerations for Publishing and Subscribing to Partial Records 17-6
Considerations When Subscribing to Character Fields 17-7
Error Handling 17-7
Message Class Built-In Functions 17-8
Message Class Methods 17-8
Clone 17-8
CopyRowset 17-9
CopyRowsetDelta 17-11
CopyRowsetDeltaOriginal 17-13
ExecuteEdits 17-16
GenXMLString 17-19
GetQueryString 17-20
GetRowset 17-20
LoadXMLString 17-22
Publish 17-22
SetEditTable 17-24
SetQueryString 17-25
SetStatus 17-26
SyncRequest 17-27
Update 17-28
Message Class Properties 17-29
ChannelName 17-29
Cookies 17-30
DefaultMessageVersion 17-31
DoNotPubToNodeName 17-31
GUID 17-31
IsActive 17-32

IsDelta 17-32
IsEditError 17-33
IsEmpty 17-34
IsLocal 17-34
MessageDetail 17-35
Name 17-35
NRId 17-35
PubID 17-35
PubNodeName 17-36
ResponseStatus 17-36
Size 17-37
SubName 17-37
SubscriptionProcessId 17-38

Chapter 18

Notification Classes

Understanding Notification Classes 18-1
Scope of the Notification Classes 18-1
Data Types of the Notification Classes 18-2
How to Import the Notification Classes 18-2
How to Create a Notification Object 18-3
Notification Classes Constructors 18-3
Notification 18-3
NotificationAddress 18-4
NotificationTemplate 18-5
WorklistEntry 18-6
Notification Class 18-7
Notification Class Import Statements 18-7
Notification Class Method 18-7
Send 18-7
Notification Class Properties 18-8
ContentType 18-8
dttmCreated 18-8
EmailReplyTo 18-8
FileNames 18-9
FileTitles 18-9
language_cd 18-9
Message 18-9
NotifyBCC 18-9
NotifyCC 18-10

NotifyFrom 18-10
NotifyGuid 18-10
NotifyTo 18-10
SourceComponent 18-11
SourceMarket 18-11
SourceMenu 18-11
Subject 18-11
Template 18-11
NotificationAddress Class 18-11
NotificationAddress Class Properties 18-12
Channel 18-12
Description 18-12
EmailId 18-12
Language 18-12
Oprid 18-13
NotificationTemplate Class 18-13
NotificationTemplate Class Methods 18-13
GetAndExpandTemplate 18-13
SetupCompVarsAndRcpts 18-14
SetupGenericVars 18-15
NotificationTemplate Class Properties 18-16
ComponentId 18-16
Instruction 18-16
Language 18-16
Market 18-16
Priority 18-16
Responses 18-17
Subject 18-17
TemplateId 18-17
TemplateType 18-17
Text 18-17
WorklistEntry Class 18-17
WorklistEntry Methods 18-18
Create 18-18
Save 18-19
Update 18-20
WorklistEntry Properties 18-21
actiondttm 18-21
busactivity 18-21
buseventname 18-21

busprocname 18-21
commentshort 18-21
do_replicate_flag 18-22
instanceid 18-22
instselecteddttm 18-22
inststatus 18-22
insttimeoutddttm 18-23
instworkeddttm 18-23
oprid 18-23
originatorid 18-23
prevoprid 18-23
syncid 18-23
timedout 18-23
transactionid 18-24
url 18-24
wl_priority 18-24
wldaystoselect 18-24
wldaystowork 18-24
worklistname 18-25
Notification Classes Examples 18-25
Creating a WorklistEntry 18-25
Updating a WorklistEntry 18-27

Chapter 19

Page Class

Understanding Page Class 19-1
Shortcut Considerations 19-1
Data Type of a Page Object 19-2
Scope of a Page Object 19-2
Page Class Built-in Function 19-2
Page Class Properties 19-2
DisplayOnly 19-2
Name 19-3
Visible 19-3

Chapter 20

PortalRegistry Classes

Understanding Portal Registry 20-1
Folders 20-2
Content References 20-3

Nodes 20-3
Security 20-4
Attributes 20-4
Additional Portal Objects 20-4
Using the PortalRegistry API 20-5
Using Security 20-5
Using Object Properties 20-6
Accessing Objects 20-6
Using PermissionValue and Cascading Permissions 20-7
Working With ValidFrom and ValidTo 20-9
Doing Error Handling 20-9
Understanding the Life-Cycle of a PortalRegistry Object 20-11
Viewing the PortalRegistry Class Hierarchy 20-12
Using Content References 20-14
UsageType 20-15
TemplateType 20-17
URLType 20-17
Nodes and URL 20-17
Naming Conventions 20-19
Deleting Content Considerations 20-19
Saving Content Considerations 20-19
Data Type of a PortalRegistry Object 20-20
Scope of a PortalRegistry Object 20-21
PortalRegistry Reference 20-21
Session Class Methods 20-21
FindPortalRegistries 20-21
GetLocalNode 20-22
GetNodes 20-23
GetPortalRegistry 20-23
GetRemoteNodes 20-24
PortalRegistry Class 20-24
PortalRegistry Class Methods 20-25
Close 20-25
CopyObject 20-26
Create 20-27
CreateRemote 20-28
Delete 20-29
DeleteHomepage 20-30
FindCRefByName 20-30
FindCRefByURL 20-31
FindFolderByName 20-32

FindPgltByName 20-33
GetAbsoluteContentURL 20-34
GetDefaultHPTabOID 20-35
GetQualifiedURL 20-36
GrantPermissionForComponent 20-36
GrantPermissionForScript 20-37
Open 20-38
PermissionListDelete 20-39
PermissionListSaveAs 20-39
RevokePermissionForComponent 20-40
RevokePermissionForScript 20-41
Save 20-42
PortalRegistry Class Properties 20-42
DefaultTemplate 20-43
Description 20-43
Favorites 20-43
Homepage 20-43
Name 20-44
NodeTemplates 20-44
OwnerId 20-44
PageletCategories 20-44
Portals 20-44
RootFolder 20-45
TabDefinitions 20-45
TemplateObject 20-45
PortalRegistry Collection 20-45
PortalRegistry Collection Methods 20-45
First 20-46
Item 20-46
Next 20-46
PortalRegistry Collection Property 20-47
Count 20-47
Node Class 20-47
Node Class Properties 20-48
ActiveNode 20-48
AppsRelease 20-48
ContentURI 20-48
DefaultPortalName 20-48
Description 20-48
IsDefault 20-49

IsLocal 20-49
Name 20-49
NodePassword 20-49
NodeType 20-50
PortalURI 20-50
ToolsRelease 20-50
Node Collection 20-50
Node Collection Methods 20-51
First 20-51
ItemByName 20-51
Next 20-52
Node Collection Property 20-52
Count 20-52
RemoteNode Collection 20-53
RemoteNode Collection Methods 20-53
First 20-53
ItemByName 20-53
Next 20-54
RemoteNode Collection Property 20-54
Count 20-55
Portal Class 20-55
Portal Class Method 20-55
Save 20-55
Portal Class Properties 20-56
HostNodeName 20-56
IsLocal 20-56
Name 20-56
Portal Collection 20-56
Portal Collection Methods 20-56
First 20-57
ItemByName 20-57
Next 20-58
Portal Collection Properties 20-58
Count 20-58
NodeTemplate Class 20-58
NodeTemplate Class Properties 20-59
DefaultTemplate 20-59
Name 20-59
TemplateObject 20-59
NodeTemplate Collection 20-59
NodeTemplate Collection Methods 20-60

DeleteItem 20-60
First 20-60
InsertItem 20-61
ItemByName 20-61
Next 20-62
NodeTemplate Collection Properties 20-62
Count 20-62
Folder Class 20-63
Folder Class Method 20-63
Save 20-63
Folder Class Properties 20-64
Attributes 20-64
Author 20-64
AuthorAccess 20-64
Authorized 20-64
CascadedPermissions 20-65
ContentRefs 20-65
CreationDate 20-65
Description 20-66
Folders 20-66
IsMobile 20-66
IsVisible 20-66
Label 20-67
Name 20-67
OwnerId 20-67
ParentName 20-67
Path 20-67
Permissions 20-68
Product 20-68
SequenceNumber 20-68
PublicAccess 20-68
ValidFrom 20-69
ValidTo 20-69
Folder Collection 20-69
Folder Collection Methods 20-69
DeleteItem 20-69
First 20-70
InsertItem 20-71
ItemByName 20-71
Next 20-72

Folder Collection Property 20-73
Count 20-73
Content Reference Class 20-73
Content Reference Class Methods 20-73
Save 20-73
Content Reference Class Properties 20-74
AbsoluteContentURL 20-74
AbsolutePortalURL 20-75
AssignedPagelets 20-75
Attributes 20-75
Author 20-75
AuthorAccess 20-75
Authorized 20-76
CascadedPermissions 20-76
ContentProvider 20-76
CreationDate 20-77
Data 20-77
Description 20-77
HtmlText 20-77
IsMobile 20-77
IsVisible 20-78
Label 20-78
Name 20-78
OwnerId 20-78
ParentName 20-79
Path 20-79
Permissions 20-79
Product 20-79
PublicAccess 20-80
QualifiedURL 20-80
RelativeURL 20-80
SequenceNumber 20-80
StorageType 20-81
Template 20-81
TemplateObject 20-82
TemplateType 20-82
URL 20-83
URLType 20-83
UsageType 20-84
ValidFrom 20-84

ValidTo 20-84
Content Reference Collection 20-85
Content Reference Collection Methods 20-85
DeleteItem 20-85
First 20-86
InsertItem 20-86
ItemByName 20-87
Next 20-88
Content Reference Collection Property 20-88
Count 20-88
AttributeValue Class 20-89
AttributeValue Class Properties 20-89
Label 20-89
Name 20-89
Translatable 20-90
Value 20-90
Attribute Collection 20-91
Attribute Collection Methods 20-91
DeleteItem 20-91
First 20-92
InsertItem 20-92
ItemByName 20-93
Next 20-94
Attribute Collection Property 20-94
Count 20-94
PermissionValue Class 20-94
PermissionValue Class Properties 20-95
Cascade 20-95
Name 20-95
PermissionValue Collection 20-95
PermissionValue Collection Methods 20-96
DeleteItem 20-96
First 20-97
InsertItem 20-97
ItemByName 20-98
Next 20-98
PermissionValue Collection Property 20-99
Count 20-99
TabDefinition Class 20-99
TabDefinition Class Methods 20-99
Save 20-100

TabDefinition Class Properties 20-100
AssignedPagelets 20-100
AvailableCategories 20-101
AvailablePagelets 20-101
Attributes 20-101
Author 20-101
AuthorAccess 20-101
ColumnLayout 20-102
CreationDate 20-102
Description 20-102
DynamicCategories 20-102
HelpID 20-102
HtmlText 20-103
IsLayoutLocked 20-103
IsRenameable 20-103
Label 20-103
LayoutBehavior 20-103
Name 20-104
OwnerId 20-104
Product 20-104
PublicAccess 20-104
QualifiedURL 20-105
SequenceNumber 20-105
StyleSheet 20-105
ValidFrom 20-105
ValidTo 20-105
TabDefinition Collection 20-106
TabDefinition Collection Methods 20-106
DeleteItem 20-106
First 20-107
InsertItem 20-107
ItemByName 20-108
Next 20-108
TabDefinition Collection Property 20-109
Count 20-109
AssignedPagelet Class 20-109
AssignedPagelet Class Properties 20-109
Column 20-110
LayoutBehavior 20-110
PageletName 20-110

Row 20-110
AssignedPagelet Collection 20-110
AssignedPagelet Collection Methods 20-111
DeleteItem 20-111
First 20-111
InsertItem 20-112
ItemByName 20-113
Next 20-114
AssignedPagelet Collection Property 20-114
Count 20-114
AvailableCategory Class 20-114
AvailableCategory Class Properties 20-115
AvailablePagelets 20-115
CategoryName 20-115
AvailableCategory Collection 20-115
AvailableCategory Collection Methods 20-115
First 20-116
ItemByName 20-116
Next 20-117
AvailableCategory Collection Property 20-117
Count 20-117
AvailablePagelet Class 20-117
AvailablePagelet Class Properties 20-118
CategoryLabel 20-118
CategoryName 20-118
Column 20-118
LayoutBehavior 20-118
PageletLabel 20-119
PageletName 20-119
Row 20-119
AvailablePagelet Collection 20-119
AvailablePagelet Collection Methods 20-120
First 20-120
ItemByName 20-120
Next 20-121
AvailablePagelet Collection Property 20-121
Count 20-121
DynamicCategory Collection 20-122
DynamicCategory Collection Methods 20-122
DeleteItem 20-122
First 20-123

InsertItem 20-123
ItemByName 20-124
Next 20-124
DynamicCategory Collection Property 20-125
Count 20-125
PageletCategory Class 20-125
PageletCategory Class Method 20-125
Save 20-126
PageletCategory Class Properties 20-126
Attributes 20-126
Author 20-127
AuthorAccess 20-127
Authorized 20-127
CascadedPermissions 20-127
CreationDate 20-128
Description 20-128
Label 20-128
Name 20-128
OwnerId 20-128
Pagelets 20-129
Permissions 20-129
Product 20-129
PublicAccess 20-129
SequenceNumber 20-130
PageletCategory Collection 20-130
PageletCategory Collection Methods 20-130
DeleteItem 20-130
First 20-131
InsertItem 20-131
ItemByName 20-132
Next 20-132
PageletCategory Collection Property 20-133
Count 20-133
Pagelet Class 20-133
Pagelet Class Method 20-133
Save 20-134
Pagelet Class Properties 20-134
Attributes 20-134
Author 20-135
AuthorAccess 20-135

Authorized 20-135
CascadedPermissions 20-135
ContentProvider 20-135
CreationDate 20-136
DefaultColumn 20-136
Description 20-136
EditPageContentProvider 20-136
EditPageQueryString 20-136
HelpID 20-137
Label 20-137
Name 20-137
OwnerId 20-137
ParentName 20-137
Permissions 20-137
Product 20-138
PublicAccess 20-138
QualifiedURL 20-138
SequenceNumber 20-138
URL 20-139
Pagelet Collection 20-139
Pagelet Collection Methods 20-139
DeleteItem 20-139
First 20-140
InsertItem 20-141
ItemByName 20-141
Next 20-142
Pagelet Collection Property 20-142
Count 20-142
UserHomepage Class 20-143
UserHomepage Class Method 20-143
Save 20-143
UserHomepage Properties 20-144
Greeting 20-144
UserId 20-144
UserTab 20-144
UserTab Class 20-144
UserTab Class Properties 20-145
ColumnLayout 20-145
Label 20-145
QualifiedURL 20-145

SelectedPagelets 20-145
SequenceNumber 20-146
TabName 20-146
UserTab Collection 20-146
UserTab Collection Methods 20-146
DeleteItem 20-146
First 20-147
InsertItem 20-147
ItemByName 20-148
Next 20-148
UserTab Collection Property 20-149
Count 20-149
SelectedPagelet Class 20-149
SelectedPagelet Class Properties 20-149
CategoryName 20-150
Column 20-150
IsMinimized 20-150
PageletName 20-150
Row 20-150
SelectedPagelet Collection 20-150
SelectedPagelet Collection Methods 20-151
DeleteItem 20-151
First 20-151
InsertItem 20-152
ItemByName 20-152
Next 20-153
SelectedPagelet Collection Property 20-153
Count 20-153
Favorite Class 20-154
Favorite Class Properties 20-154
CRefName 20-154
Label 20-154
QualifiedURL 20-154
SequenceNumber 20-155
URL 20-155
Favorite Collection 20-155
Favorite Collection Methods 20-155
DeleteItem 20-155
First 20-156
InsertItem 20-157
ItemByLabel 20-157

Next 20-158
Save 20-158
Favorite Collection Property 20-159
Count 20-159
PortalRegistry Classes Example 20-159
Changing PortalRegistry Properties 20-159
Adding a ContentProvider 20-160
Changing ContentProvider Properties 20-163
Adding a Folder 20-166
Adding a Content Reference 20-169
Setting Permissions Using the PermissionValue Object 20-172
Using Attributes 20-174
Visual Basic Example 20-175
C/C++ Example 20-176

Chapter 21

ProcessRequest Class

Understanding ProcessRequest Class 21-1
Data Type of a ProcessRequest Object 21-2
Scope of a ProcessRequest Object 21-2
Options for Items In Jobs and Jobsets 21-2
Values for Output Type and Format 21-5
Alternative Option To Specify Email or Web Attributes 21-7
ProcessRequest Class Built-in Function 21-8
ProcessRequest Class Methods 21-8
AddDistributionOption 21-9
PrintJobHTMLRpt 21-10
PrintJobRqstRpt 21-11
PrintSchdlHTMLRpt 21-14
RunJobSetNow 21-16
Schedule 21-17
SetEmailOption 21-18
SetItemFolder 21-19
SetOutputOption 21-20
UpdateRunStatus 21-21
ProcessRequest Class Properties 21-22
EmailAttachLog 21-22
EmailSubject 21-22
EmailText 21-22
EmailWebReport 21-23

JobName 21-23
LanguageCd 21-23
OutDest 21-24
OutDestFormat 21-25
OutDestType 21-25
PortalFolder 21-26
ProcessInstance 21-26
ProcessName 21-26
ProcessType 21-27
RunControlID 21-28
RunDateTime 21-28
RunLocation 21-29
RunRecurrence 21-29
RunStatus 21-29
Status 21-30
TimeZone 21-31
ProcessRequest Class Examples 21-31
Scheduling a Single Process 21-31
Scheduling a Job Where Job Item Changes Are Not Made 21-32
Scheduling a Job Where Job Item Changes Are Made 21-32

Chapter 22

Query Classes

Understanding Query Classes 22-1
Collections in the Query Classes 22-1
Life-Cycle of a Query 22-2
Query Class Hierarchy 22-3
Query Classes Overview 22-5
Query 22-6
QuerySelect 22-6
QueryRecords 22-6
QueryOutputFields 22-6
QuerySelectedFields 22-7
QueryCriteria 22-7
QueryDBRecords and QueryDBRecordFields 22-7
Working With Criteria and Expressions 22-7
Setting the Type 22-8
Considerations Adding New Expressions 22-8
Operator and Expression 2 Dependencies 22-9
Query Monitor 22-9

Using Query Metadata 22-10
Running a Query 22-11
Specifying User's Language 22-12
Query Security 22-12
Error Handling With Query Classes 22-13
Understanding QueryOutputFields and QuerySelectedFields 22-14
Understanding Having Criteria 22-14
Data Type of Query Objects 22-15
Scope of the Query Objects 22-15
Session Class Methods 22-16
FindQueryDBRecords 22-16
FindQueries 22-17
FindQueriesDateRange 22-17
GetQuery 22-18
GetQuerySecurityProfile 22-19
SearchQueryDBRecords 22-20
SearchPrivateQueries 22-21
SearchPublicQueries 22-22
Query Collection 22-24
Query Collection Methods 22-24
First 22-24
Item 22-25
ItemByName 22-26
Next 22-26
Query Collection Property 22-27
Count 22-27
Query Class 22-27
Query Class Methods 22-27
AddQuerySelect 22-27
AddPrompt 22-28
Close 22-28
Create 22-29
Delete 22-31
DeletePrompt 22-31
FindExpression 22-32
FormatResultString 22-32
Open 22-33
Rename 22-34
RunToFile 22-35
RunToRowset 22-38

Save 22-41
Query Class Properties 22-41
Approved 22-41
ApproveOprId 22-42
ApproveUserId 22-42
ApproveDtTm 22-42
CreateOprId 22-42
CreateDtTm 22-43
CreateUserId 22-43
Description 22-43
ExecAppName 22-43
ExecLogging 22-43
LastSQLErrorCode 22-44
LastUpdDttm 22-44
LastUpdOprId 22-44
LongDescription 22-44
Metadata 22-44
Name 22-45
PDFFont 22-45
Prompts 22-45
PromptRecord 22-46
PublicPrivate 22-46
QuerySelect 22-46
QueryStatistics 22-47
RunTimePrompts 22-47
SQL 22-47
Type 22-47
QuerySelect Class 22-48
QuerySelect Methods 22-49
AddAllFields 22-49
AddCriteria 22-50
AddExpression 22-50
AddHavingCriteria 22-51
AddQueryExpressionField 22-51
AddQueryOutputField 22-52
AddQueryRecord 22-53
AddQuerySelectedField 22-53
DeleteCriteria 22-54
DeleteExpression 22-54
DeleteField 22-55

DeleteHavingCriteria 22-56
DeleteRecord 22-56
QuerySelect Properties 22-57
Criteria 22-57
Distinct 22-57
Expressions 22-57
HavingCriteria 22-57
ParentSelectNum 22-58
QueryOutputFields 22-58
QueryRecords 22-58
QuerySelectedFields 22-58
QuerySelects 22-59
SelectNum 22-59
SelectType 22-59
QuerySelect Collection 22-59
QuerySelect Collection Methods 22-60
AddUnion 22-60
DeleteQuerySelect 22-60
First 22-61
Item 22-61
ItemBySelNum 22-62
Next 22-63
QuerySelect Collection Property 22-63
Count 22-63
QueryRecord Collection 22-63
QueryRecord Collection Methods 22-64
First 22-64
Item 22-64
ItemByAlias 22-65
Next 22-65
QueryRecord Collection Property 22-66
Count 22-66
QueryRecord Class 22-66
QueryRecord Class Method 22-66
GetField 22-67
QueryRecord Class Properties 22-67
Description 22-67
JoinAlias 22-67
JoinFieldName 22-68
JoinType 22-68

Name 22-68
QueryFields 22-68
RecordAlias 22-69
QueryField Collection 22-69
QueryField Collection Methods 22-69
First 22-70
Item 22-70
ItemByNameAndAlias 22-71
ItemByExpNum 22-71
Next 22-72
QueryField Collection Property 22-72
Count 22-72
QueryField Class 22-72
QueryField Methods 22-73
AddTranslateExpression 22-73
AddTranslateField 22-73
QueryField Class Properties 22-74
Aggregate 22-74
ColumnNumber 22-75
Description 22-75
Decimal 22-75
ExpNum 22-75
Flag 22-75
Format 22-75
HeadingText 22-76
HeadingType 22-76
HeadingUniqueFieldName 22-76
Length 22-77
LongName 22-77
Name 22-77
OrderByDirection 22-77
OrderByNumber 22-78
QueryRecord 22-78
RecordAlias 22-78
ShortName 22-79
TranslateEffDtLogic 22-79
TranslateExpression 22-79
TranslateField 22-79
TranslateOption 22-80
Type 22-80

QueryCriteria Collection 22-80
QueryCriteria Collection Methods 22-81
First 22-81
Item 22-81
ItemByName 22-82
Next 22-82
QueryCriteria Collection Property 22-83
Count 22-83
QueryCriteria Class 22-83
QueryCriteria Class Methods 22-83
AddExpr1Expression 22-84
AddExpr1Field 22-84
AddExpr2Expression 22-85
AddExpr2Field1 22-85
AddExpr2Field2 22-86
AddExpr2List 22-87
AddExpr2Subquery 22-87
QueryCriteria Class Properties 22-88
Expr1Expression 22-88
Expr1Field 22-88
Expr1Type 22-88
Expr2Constant1 22-89
Expr2Constant2 22-90
Expr2Expression1 22-90
Expr2Expression2 22-90
Expr2Field1 22-90
Expr2Field2 22-91
Expr2List 22-91
Expr2Subquery 22-91
Expr2Type 22-92
Logical 22-95
LParenLvl 22-95
Name 22-95
Negation 22-95
Operator 22-96
R1ExprNum 22-97
R2ExprNum 22-98
R1ExprType 22-98
R2ExprType 22-98
RParenLvl 22-98

QueryExpression Collection 22-98
QueryExpression Collection Methods 22-99
First 22-99
Item 22-99
ItemByName 22-100
Next 22-100
QueryExpression Collection Property 22-101
Count 22-101
QueryExpression Class 22-101
QueryExpression Class Properties 22-101
Aggregate 22-102
BindFlag 22-102
Decimal 22-102
ExpNum 22-102
IsXlatExpression 22-102
Length 22-103
Name 22-103
OutputField 22-103
RightExprFlag 22-103
SelectedField 22-103
Text 22-104
Type 22-104
QueryList Class 22-104
QueryList Class Methods 22-105
AddListValue 22-105
First 22-105
Item 22-106
Next 22-106
QueryList Class Property 22-107
Count 22-107
QueryListValue Class 22-107
QueryListValue Class Methods 22-107
QueryListValue Class Properties 22-107
IsPrompt 22-108
Value 22-108
QueryRecordHierarchy Collection 22-108
QueryRecordHierarchy Collection Methods 22-108
First 22-108
Item 22-109
ItemByName 22-109

Next 22-110
QueryRecordHierarchy Collection Property 22-110
Count 22-110
QueryRecordHierarchy Class 22-111
QueryRecordHierarchy Class Properties 22-111
Description 22-111
Level 22-111
Name 22-111
ParentFlag 22-112
Query Metadata Collection 22-112
Query Metadata Collection Methods 22-112
First 22-112
Item 22-113
ItemByName 22-113
Next 22-114
Query Metadata Collection Property 22-114
Count 22-114
Query Metadata Class 22-115
Query Metadata Class Properties 22-115
Name 22-115
Value 22-116
QueryStatistics Class 22-116
QueryStatistics Methods 22-116
QueryStatistics Class Properties 22-116
AvgExecTime 22-116
AvgFetchTime 22-116
AvgNumRows 22-116
ExecCount 22-117
LastExecDtTm 22-117
QuerySecurityProfile Class 22-117
QuerySecurityProfile Class Methods 22-117
QuerySecurityProfile Class Properties 22-117
AllowAnyJoin 22-117
AllowDistinct 22-118
AllowExpressions 22-118
AllowSubqueries 22-118
AllowUnions 22-118
ApprovePrivateQuery 22-118
ApprovePublicQuery 22-118
CanCreatePublic 22-119
CanCreateWorkFlow 22-119

CanModifyQuery 22-119
CanRunQuery 22-119
CanRunToCrystal 22-119
CanRunToExcel 22-119
LimitUnapproved 22-120
MaxInTreeCriteria 22-120
MaxJoins 22-120
MaxRowsToFetch 22-120
MaxUnapprovedRows 22-120
QueryDBRecord Collection 22-121
QueryDBRecord Collection Methods 22-121
First 22-121
Item 22-121
ItemByName 22-122
Next 22-122
QueryDBRecord Collection Property 22-123
Count 22-123
QueryDBRecord Class 22-123
QueryDBRecord Class Methods 22-123
QueryDBRecordFieldByIndex 22-124
QueryDBRecordFieldByName 22-124
QueryDBRecord Class Properties 22-125
Description 22-125
Name 22-125
QueryDBRecordFields 22-125
RecordHierachy 22-125
QueryDBRecordField Collection 22-126
QueryDBRecordField Collection Methods 22-126
First 22-126
Item 22-126
ItemByName 22-127
Next 22-128
Sort 22-128
QueryDBRecordField Collection Property 22-129
Count 22-129
QueryDBRecordField Class 22-129
QueryDBRecordField Class Properties 22-129
Decimal 22-130
Description 22-130
Flag 22-130

Format 22-131
Length 22-132
LongName 22-132
LookupTableName 22-132
LookupTableRecord 22-132
Name 22-132
Shortname 22-132
Type 22-132
QueryPrompt Collection 22-133
QueryPrompt Collection Methods 22-133
First 22-133
Item 22-134
ItemByName 22-134
Next 22-135
QueryPrompt Collection Property 22-135
Count 22-135
QueryPrompt Class 22-136
QueryPrompt Properties 22-136
EditType 22-136
FieldDecimal 22-136
FieldFormat 22-137
FieldLength 22-137
FieldName 22-137
FieldType 22-138
HeadingText 22-138
HeadingType 22-138
LangCount 22-139
Name 22-139
PromptRecordFieldName 22-139
PromptTable 22-139
UniquePromptName 22-139
UseCount 22-139
Query Classes Examples 22-140
Creating a New Query 22-140
Adding Criteria 22-142

Chapter 23

Record Class

Understanding Record Class 23-1
Shortcut Considerations 23-2

Record Methods Used to Create SQL Statements 23-2
Data Type of a Record Object 23-3
Scope of a Record Object 23-3
Record Class Built-in Functions 23-3
Record Class Methods 23-3
CompareFields 23-4
CopyChangedFieldsTo 23-4
CopyFieldsTo 23-5
DBPatternMatch 23-7
Delete 23-7
ExecuteEdits 23-9
GetField 23-11
Insert 23-13
SearchClear 23-15
SelectByKey 23-16
SelectByKeyEffDt 23-17
SetDefault 23-18
SetEditTable 23-19
Update 23-21
Record Class Properties 23-23
FieldCount 23-23
fieldname 23-23
IsChanged 23-24
IsDeleted 23-24
IsEditError 23-25
Name 23-25
ParentRow 23-26
RelLangRecName 23-26

Chapter 24

Row Class

Understanding Row Class 24-1
Shortcut Considerations 24-2
Data Type of a Row Object 24-2
Scope of a Row Object 24-2
Row Class Built-in Function 24-2
Row Class Methods 24-2
CopyTo 24-3
GetNextEffRow 24-3
GetPriorEffRow 24-4

GetRecord 24-5
GetRowset 24-6
scrollname 24-7
Row Class Properties 24-8
ChildCount 24-8
DeleteEnabled 24-8
IsChanged 24-9
IsDeleted 24-10
IsEditError 24-10
IsNew 24-11
ParentRowset 24-11
recname 24-12
RecordCount 24-12
RowNumber 24-12
Selected 24-13
Style 24-13
Visible 24-14

Chapter 25

Rowset Class

Understanding Rowset Class 25-1
Shortcut Considerations 25-2
Data Type of a Rowset Object 25-2
Scope of a Rowset Object 25-2
Rowset Class Built-in Functions 25-3
Rowset Class Methods 25-3
ClearDeletesChanges 25-3
CopyTo 25-5
DeleteRow 25-7
Fill 25-8
FillAppend 25-11
Flush 25-13
FlushRow 25-14
GetCurrEffRow 25-14
GetNewEffRow 25-15
GetRow 25-16
HideAllRows 25-17
InsertRow 25-18
Refresh 25-19
Select 25-20

SelectNew 25-22
SetDefault 25-24
ShowAllRows 25-25
Sort 25-26
Rowset Class Properties 25-27
ActiveRowCount 25-27
ChangeOnInit 25-28
DataAreaCollapsed 25-28
DBRecordName 25-29
DeleteEnabled 25-29
EffDt 25-30
EffSeq 25-31
InsertEnabled 25-31
IsEditError 25-32
Level 25-33
Name 25-33
ParentRow 25-33
ParentRowset 25-34
RowCount 25-34
TopRowNumber 25-34

Chapter 26

Search Classes

Understanding Search Class 26-1
Using the SearchResults Collection 26-2
Understanding SearchResults 26-3
Search Classes Hierarchy 26-4
Error Handling 26-5
Data Type of Search Objects 26-6
Scope of the Search Objects 26-6
Session Class Methods 26-6
GetSearchIndexes 26-7
GetSearchQuery 26-7
PortalRegistry Class Methods 26-8
BuildSearchIndex 26-8
GetSearchQuery 26-9
SearchQuery Class 26-9
SearchQuery Class Method 26-10
Execute 26-10
SearchQuery Class Properties 26-10

HitCount 26-10
IndexName 26-11
KnowledgeBase 26-11
Language 26-11
ProcessedCount 26-11
QueryText 26-11
SearchResult Collection 26-12
SearchResult Collection Methods 26-12
First 26-12
Item 26-12
Next 26-13
SearchResult Collection Property 26-13
Count 26-13
SearchResult Class 26-13
SearchResult Class Properties 26-14
Key 26-14
Score 26-14
SearchFields 26-14
SearchField Collection 26-14
SearchField Collection Methods 26-15
First 26-15
ItemByName 26-15
Next 26-16
SearchField Collection Property 26-16
Count 26-16
SearchField Class 26-16
SearchField Class Properties 26-16
Name 26-17
Value 26-17
SearchIndexes Collection 26-17
SearchIndexes Collection Methods 26-17
First 26-17
ItemByName 26-18
Next 26-18
SearchIndex Collection Property 26-18
Count 26-19
SearchIndex Class 26-19
SearchIndex Class Method 26-19
Save 26-19
SearchIndex Class Properties 26-20
ExtraOptions 26-20

FSOpts 26-20
HTTPOpts 26-20
Languages 26-20
Location 26-21
Name 26-21
RecOpts 26-21
Schedules 26-21
Type 26-21
SearchRecord Options Class 26-22
SearchRecord Options Class Properties 26-22
Fields 26-22
Filter 26-23
IncrementalView 26-23
RecName 26-23
VeggieKey 26-23
ZoneOptions 26-24
SearchRecordField Collection 26-25
SearchRecordField Collection Methods 26-25
DeleteItem 26-25
First 26-25
InsertItem 26-26
ItemByName 26-26
Next 26-27
SearchRecordField Collection Property 26-27
Count 26-27
SearchRecordField Class 26-27
SearchRecordField Class Properties 26-28
FieldName 26-28
IsAttachment 26-28
IsVerityField 26-28
IsWordIndex 26-28
RecordName 26-29
Search Language Collection 26-29
Search Language Collection Methods 26-29
DeleteItem 26-29
First 26-30
InsertItem 26-30
ItemByName 26-30
Next 26-31
Search Language Collection Property 26-31

Count 26-31
Search Language Class 26-32
Search Language Class Properties 26-32
LanguageCd 26-32
MapLanguageCd 26-32
Search Schedule Collection 26-32
Search Schedule Collection Methods 26-32
DeleteItem 26-33
First 26-33
InsertItem 26-33
ItemByName 26-34
Next 26-34
Search Schedule Collection Property 26-35
Count 26-35
Search Schedule Class 26-35
Search Schedule Class Properties 26-35
BuildType 26-35
RunCntlId 26-36
RunRecurrence 26-36
ServerName 26-36
Search HTTP Options Class 26-36
Search HTTP Options Class Properties 26-37
AllowHTTPS 26-37
DomainLimit 26-37
GlobList 26-37
GlobListType 26-38
LinkDepth 26-38
MIMEList 26-39
MIMEListType 26-39
ProxyHost 26-39
ProxyPort 26-39
StartOpts 26-39
Search FS Options Class 26-40
Search FS Options Class Properties 26-40
GlobList 26-40
GlobListType 26-40
MIMEList 26-41
MIMEListType 26-41
StartOpts 26-41
Search Start Options Collection 26-42
Search Start Options Collection Methods 26-42

DeleteItem 26-42
First 26-43
InsertItem 26-43
ItemByName 26-44
Next 26-44
Search Start Options Collection Property 26-44
Count 26-45
Search Start Options Class 26-45
Search Start Options Class Properties 26-45
IsDomainRestricted 26-45
IsHostRestricted 26-45
Value 26-46
Search API Examples 26-46
General Purpose Search API Example 26-46
Portal Search API Example 26-47

Chapter 27

Session Class

Understanding Session Class 27-1
Security and Access to the PeopleSoft System 27-2
Error Handling 27-3
Displaying Error Messages 27-4
Regional Settings 27-4
Trace Settings 27-4
Session Object Declaration 27-6
State Considerations 27-6
Considerations for Declaring Collections 27-7
Scope of a Session Object 27-7
Session Class Built-in Function 27-7
Session Class Methods 27-7
Connect 27-7
Disconnect 27-9
API Instantiation Methods 27-9
Session Class Properties 27-11
ErrorPending 27-11
PSMessages 27-11
PSMessagesMode 27-12
RegionalSettings 27-12
Repository 27-13
SuspendFormatting 27-13

TraceSettings 27-13
WarningPending 27-13
Accessing a PSMessages Collection 27-14
PSMessages Collection Methods 27-14
DeleteAll 27-14
DeleteItem 27-15
First 27-15
Item 27-16
Next 27-16
PSMessages Collection Property 27-16
Count 27-17
PSMessage Class Properties 27-17
ExplainText 27-17
MessageNumber 27-17
MessageSetNumber 27-17
MessageType 27-18
Source 27-18
Text 27-20
Regional Settings Class Properties 27-21
ClientTimeZone 27-21
CurrencyFormat 27-21
CurrencySymbol 27-22
DateFormat 27-22
DateSeparator 27-22
DecimalSymbol 27-22
DigitGroupingSymbol 27-23
LanguageCD 27-23
UseLocalTime 27-24
1159Separator 27-24
2359Separator 27-24
Trace Setting Class Properties 27-24
API 27-25
COBOLStmtTimings 27-25
ConnDisRollbackCommit 27-25
DBSpecificCalls 27-25
ManagerInfo 27-25
NetworkServices 27-25
NonSSBs 27-26
OutputUNICODE 27-26
PCExtFcnCalls 27-26

PCFcnReturnValues 27-26
PCFetchedValues 27-26
PCIntFcnCalls 27-27
PCListProgram 27-27
PCParameterValues 27-27
PCProgramStatements 27-27
PCStack 27-27
PCStartOfPrograms 27-27
PCTraceProgram 27-28
PCVariableAssignments 27-28
RowFetch 27-28
SQLStatement 27-28
SQLStatementVariables 27-28
SSBs 27-29
SybBindInfo 27-29
SybFetchInfo 27-29
TraceFile 27-29

Chapter 28

SOAPDoc Class

Understanding SOAPDoc Class 28-1
SOAP Message Exchange Model 28-2
SOAP Message Document 28-2
The SOAPDoc Class 28-4
SOAPDoc Object Creation 28-4
SOAP Header Section 28-4
SOAP Envelope Section 28-5
SOAP Body Section 28-5
SOAP Method Section 28-5
SOAPDoc Section Access 28-6
PeopleCode to Create a SOAPDoc 28-6
SOAPDoc Fault Section 28-7
The ValidateSOAPDoc Method 28-7
SOAPDoc Objects and Messages 28-8
Data Type of a SOAPDoc Object 28-8
Scope of a SOAPDoc Object 28-8
SOAPDoc Built-in Functions 28-8
SOAPDoc Class Methods 28-9
AddBody 28-9
AddEnvelope 28-10

AddFault 28-11
AddMethod 28-13
AddParm 28-14
GetParmName 28-15
GetParmValue 28-15
ValidateSOAPDoc 28-16
SOAPDoc Class Properties 28-17
BodyNode 28-17
EnvelopeNode 28-18
FaultCode 28-18
FaultString 28-18
MethodName 28-18
MethodNode 28-19
ParmCount 28-19
XmlDoc 28-19
SOAPDoc Class Examples 28-20
Creating a SOAP Document 28-20
Reading an Existing SOAP Document 28-21
Using SOAP XML Text 28-21

Chapter 29

SQL Class

Understanding SQL Class 29-1
Using Record Class SQL 29-3
Creating a SQL Definition 29-3
Binding and Executing of SQL Statements 29-3
Fetching From a Select 29-4
Using Array of Any for Bind Values or Fetch Results 29-5
Understanding Different "Styles" of SELECT 29-5
Reusing a Cursor 29-6
Using the ReuseCursor Property 29-6
Reusing a Cursor Without the ReuseCursor Property 29-7
Understanding SQL Objects and PeopleSoft Application Engine Programs 29-9
Declaring a SQL Object 29-9
Scope of an SQL Object 29-9
SQL Class Built-in Functions 29-10
SQL Class Methods 29-10
Close 29-10
Execute 29-11
Fetch 29-13

Open 29-15
SQL Class Properties 29-16
BulkMode 29-16
IsOpen 29-17
LTrim 29-18
ReuseCursor 29-19
RowsAffected 29-19
Status 29-20
TraceName 29-20
Value 29-21

Chapter 30

Tree Classes

Understanding Tree Classes 30-1
Relationships Between Different Tree Classes 30-2
Collections in the Tree Classes 30-4
Error Handling With Trees 30-5
Leaves and Nodes Insert Verification 30-7
Data Types for Tree Objects 30-7
Scope of the Tree Objects 30-7
Tree Classes Implementation 30-8
Session Class Methods 30-11
FindTree 30-11
FindTreeStructure 30-13
GetTree 30-14
GetTreeStructure 30-15
Branch Collection 30-15
Branch Collection Method 30-16
Item 30-16
Branch Collection Properties 30-17
Count 30-17
First 30-17
Last 30-17
Next 30-17
Leaf Class 30-17
Leaf Class Methods 30-18
Cut 30-18
Delete 30-18
DeleteByRange 30-19
InsertDynSib 30-19

InsertSib 30-20
MoveAsChild 30-21
MoveAsChildByName 30-22
MoveAsSib 30-22
MoveAsSibByRange 30-23
PasteSib 30-24
Leaf Class Properties 30-24
DisplayLevelNumber 30-24
Dynamic 30-25
HasNextSib 30-25
HasPrevSib 30-25
IsChanged 30-25
IsCut 30-25
IsDeleted 30-26
IsInserted 30-26
NextSib 30-26
Parent 30-26
PrevSib 30-27
RangeFrom 30-27
RangeTo 30-27
TreeBranchName 30-27
TreeEffDt 30-27
TreeName 30-28
TreeSetId 30-28
TreeUserKeyValue 30-28
Level Collection 30-28
LevelCollection Methods 30-28
Add 30-28
Item 30-29
Remove 30-29
Save 30-30
Level Collection Properties 30-30
Count 30-30
First 30-30
Last 30-30
Next 30-31
Level Class 30-31
Level Class Methods 30-31
Create 30-31
Delete 30-32

Level Class Properties 30-32
AllValuesAudit 30-32
Description 30-32
Name 30-33
Number 30-33
TreeBranchName 30-33
TreeEffDt 30-33
TreeName 30-33
TreeSetId 30-33
TreeUserKeyValue 30-33
Node Class 30-34
Node Class Methods 30-34
Branch 30-34
Cut 30-35
Delete 30-35
DeleteByName 30-35
Expand 30-36
InsertChildLeaf 30-36
InsertChildNode 30-37
InsertChildRecord 30-38
InsertDynChildLeaf 30-39
InsertSib 30-40
InsertSibRecord 30-40
MoveAsChild 30-41
MoveAsChildByName 30-42
MoveAsSib 30-42
MoveAsSibByName 30-43
PasteChild 30-44
PasteSib 30-44
RefreshDescription 30-45
Rename 30-45
SwitchLevel 30-46
Unbranch 30-46
Node Class Properties 30-47
AllChildCount 30-47
AllChildNodeCount 30-47
ChildLeafCount 30-47
ChildNodeCount 30-48
Description 30-48
DisplayLevelNumber 30-48

FirstChildLeaf 30-48
FirstChildNode 30-48
HasChildLeaves 30-49
HasChildNodes 30-49
HasChildren 30-49
HasNextSib 30-49
HasPrevSib 30-49
IsBranched 30-50
IsChanged 30-50
IsCut 30-50
IsDeleted 30-50
IsInserted 30-50
IsRoot 30-50
LastChildLeaf 30-51
LastChildNode 30-51
LevelNumber 30-51
Name 30-51
NextSib 30-52
Parent 30-52
PrevSib 30-52
State 30-52
TreeBranchName 30-53
TreeEffDt 30-53
TreeName 30-53
TreeSetId 30-53
TreeUserKeyValue 30-54
Type 30-54
Tree Class 30-54
Tree Class Methods 30-55
Audit 30-55
AuditByName 30-55
Close 30-57
Copy 30-57
Create 30-59
Delete 30-60
Exists 30-61
FindLeaf 30-62
FindNextDisplayObject 30-63
FindNode 30-64
FindPrevDisplayObject 30-66

FindRoot 30-67
InsertRoot 30-67
LeafExists 30-68
Open 30-69
OpenWholeTree 30-70
NodeExists 30-71
Rename 30-72
Save 30-73
SaveAs 30-74
SaveAsDraft 30-75
SaveDraft 30-77
Tree Class Properties 30-77
AllValues 30-77
AuditDetails 30-78
Branches 30-78
BranchLevel 30-78
BranchName 30-79
Category 30-79
CheckLeafUserData 30-79
Description 30-79
DisplayLeaf 30-80
DisplayNode 30-80
DuplicateLeaves 30-80
EffDt 30-80
HasDetailRanges 30-81
IsBranched 30-81
IsChanged 30-81
IsOpen 30-82
IsQueryTree 30-82
IsReadOnly 30-82
IsValid 30-83
IsWholeTree 30-83
KeyBranchName 30-83
KeyEffDt 30-83
KeyName 30-84
KeySetId 30-84
KeyUserKeyValue 30-84
LeafCount 30-84
LeafOnClipboard 30-85
LevelCount 30-85

Levels 30-85
LevelUse 30-85
Name 30-86
NodeCount 30-86
NodeOnClipboard 30-87
ParentLevel 30-87
ParentName 30-87
PerformanceMethod 30-87
PerformanceSelector 30-88
PerformanceSelectorOption 30-88
SetID 30-89
Status 30-89
Structure 30-89
StructureName 30-90
TotalNodeCount 30-90
UserKeyValue 30-90
Tree Collection 30-91
Tree Collection Method 30-92
Item 30-92
Tree Collection Properties 30-93
Count 30-94
First 30-94
Last 30-94
Next 30-94
Tree Structure Class 30-94
Tree Structure Class Methods 30-95
Close 30-95
Copy 30-95
Create 30-96
Delete 30-96
Open 30-97
Rename 30-97
Save 30-98
Tree Structure Class Properties 30-98
Description 30-98
DetailComponent 30-98
DetailField 30-99
DetailMenu 30-99
DetailMenuBar 30-100
DetailMenuItem 30-100

DetailMultiNavigate 30-100
DetailPage 30-100
DetailRecord 30-101
IndirectionMethod 30-101
KeyName 30-103
LevelComponent 30-103
LevelMenu 30-104
LevelMenuBar 30-104
LevelMenuItem 30-104
LevelPage 30-104
LevelRecord 30-105
Name 30-105
NodeComponent 30-105
NodeField 30-106
NodeMenu 30-106
NodeMenuBar 30-106
NodeMenuItem 30-106
NodeMultiNavigate 30-107
NodePage 30-107
NodeRecord 30-107
NodeUserKeyField 30-108
SummarySetId 30-108
SummaryLevelNumber 30-108
SummaryTreeName 30-108
SummaryUserKeyValue 30-109
Type 30-109
Tree Structure Collection 30-109
Tree Structure Collection Method 30-110
Item 30-110
Tree Structure Collection Properties 30-111
Count 30-111
First 30-111
Last 30-111
Next 30-111
Traverse Tree Hierarchy Example 30-111

Chapter 31

XmlDoc Class

Understanding XmlDoc Class 31-1
When to Use an XmlDoc Object 31-2

XmlDoc Object Creation 31-3
XmlNode Class Considerations 31-3
Accessing and Traversing an XmlNode Object 31-4
Error Handling 31-4
SOAPDoc Object Considerations 31-5
Scope of XmlDoc and XmlNode Objects 31-5
Data Type of an XmlDoc or XmlNode Object 31-5
XmlDoc Class Built-in Functions 31-6
XmlDoc Methods 31-6
CopyRowset 31-6
CopyPSFTMessage 31-7
CopyToRowset 31-9
CreateDocumentElement 31-11
CreateDocumentType 31-12
GenXmlString 31-13
GenFormattedXmlString 31-14
GetElementsByTagName 31-14
ParseXmlFromURL 31-15
ParseXmlString 31-16
XmlDoc Property 31-17
DocumentElement 31-17
XmlNode 31-17
XmlNode Methods 31-17
AddAttribute 31-17
AddAttributeNS 31-19
AddCDataSection 31-20
AddComment 31-20
AddElement 31-21
AddEntityReference 31-22
AddNode 31-23
AddProcessInstruction 31-23
AddText 31-24
CopyNode 31-25
FindNode 31-26
FindNodes 31-27
GenXmlString 31-28
GetCDataValue 31-28
GetCDataValues 31-29
GetChildNode 31-29
GetElement 31-30

GetElements 31-30
GetElementsByTagName 31-31
InsertAttribute 31-32
InsertAttributeNS 31-32
InsertCDataSection 31-33
InsertComment 31-34
InsertElement 31-34
InsertEntityReference 31-35
InsertNode 31-35
InsertProcessInstruction 31-36
InsertText 31-37
RemoveChildNode 31-37
RemoveAllChildNode 31-38
XmlNode Properties 31-38
AttributeCount 31-38
ChildNodeCount 31-39
Index 31-39
IsNull 31-39
LocalName 31-39
NamespaceURI 31-39
NextSibling 31-40
NodeName 31-40
NodePath 31-40
NodeType 31-40
NodeValue 31-41
ParentNode 31-41
Prefix 31-41
PreviousSibling 31-41

Chapter 32

Appendix A: Cheat Sheets

Class List Cheat Sheet 32-1
Summary of PeopleCode Classes 32-1
Session Class Methods and Properties 32-23
Component Interface Class Methods and Properties 32-25
Search Classes Methods and Properties 32-28
PortalRegistry Classes Methods and Properties 32-32
Query Classes Methods and Properties 32-45
Tree Classes Methods and Properties 32-56
Internet Script Classes Methods and Properties 32-63

Mapping of Functions to Methods and Properties 32-65
Mapping of Old Names to New Names 32-67
Deprecated Functions 32-68
PeopleCode Syntax Cheat Sheet 32-68

Chapter 33

Appendix B: Viewing Trees From Application Pages
Understanding View Trees 33-1
Invoking View Trees From Application Pages 33-3
View Trees From Application Example–Without Multi-Node Selection (Method A) 33-6
View Trees From Application Example–With Multi-Node Selection (Method B) 33-7

Glossary

Index

PeopleCode Reference Preface

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 This PeopleBook" section contains general product line information, such as related documentation, common page elements, and typographical conventions. This book also contains a glossary with useful terms that are used in PeopleBooks.

See **PeopleSoft Glossary**.

See also *PeopleCode Developer's Guide*

About This PeopleBook

This book provides you with the information that you need for implementing and using *PeopleTools 8.4* applications. Complete documentation for this release is provided on the CD-ROM PT84PBR0.

Note. Your access to PeopleSoft PeopleBooks depends on which PeopleSoft applications you've licensed. You may not have access to all of the PeopleBooks.

This section contains information that you should know before you begin working with PeopleSoft products and documentation, including PeopleSoft-specific documentation conventions, information specific to each PeopleSoft product line, and information on ordering additional copies of our documentation.

Before You Begin

To benefit fully from the information covered in this book, you should have a basic understanding of how to use PeopleSoft applications. We recommend that you 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.

Because we assume that you already know how to navigate the PeopleSoft system, much of the information in these books is not procedural. That is, these books do not typically provide

step-by-step instructions on using tables, pages, and menus. Instead, we provide you with the information that you need to use the system most effectively and to implement your PeopleSoft application according to your organizational or departmental needs. PeopleBooks expand on the material covered in PeopleSoft training classes.

PeopleSoft Application Fundamentals

Each PeopleSoft application PeopleBook provides implementation and processing information for your PeopleSoft database. However, there is additional, essential information describing the setup and design of your database contained in a companion volume of documentation called *PeopleSoft Application Fundamentals.*

PeopleSoft Application Fundamentals contains important topics that apply to many or all PeopleSoft applications across each product line. Whether you are implementing only one PeopleSoft application, some combination of products within a product line, or an entire PeopleSoft system, you should be familiar with the contents of this central PeopleBook. It contains fundamental information such as setting up control tables and administering security.

The PeopleSoft Applications Fundamentals PeopleBook contains common information pertinent to all applications in each product line, such as defining general options. If you're upgrading from a previous PeopleSoft release, you may notice that we've removed some topics or topic headings from the individual application PeopleBooks and consolidated them in this single reference book. You'll now find only application-specific information in your individual application PeopleBooks. This makes the documentation as a whole less redundant. Throughout each PeopleBook, we provide cross-references to *PeopleSoft Application Fundamentals* and other PeopleBooks.

Related Documentation

You can order printed, bound versions of the complete PeopleSoft documentation delivered on your PeopleBooks CD-ROM and additional copies of the PeopleBooks CDs through the Documentation section of the PeopleSoft Customer Connection website: http://www.peoplesoft.com/corp/en/login.asp

You can find updates and additional documentation for this release, as well as previous releases, on PeopleSoft Customer Connection (http://www.peoplesoft.com/corp/en/login.asp). Through the Documentation section of 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 delivered on your PeopleBooks CD.

Important! Before you upgrade, it is *imperative* that you check PeopleSoft Customer Connection for updates to the upgrade instructions. We continually post updates as we refine the upgrade process.

Hard-copy Documentation

To order printed, bound volumes of the complete PeopleSoft documentation delivered on your PeopleBooks CD-ROM, visit the PeopleSoft Press website from the Documentation section of PeopleSoft Customer Connection. The PeopleSoft Press website is a joint venture between PeopleSoft and Consolidated Publications Incorporated (CPI), our book print vendor.

We make 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 the following methods:

Internet	From the main PeopleSoft Internet site, go to the Documentation section of Customer Connection. You can find order information under the Ordering PeopleBooks topic. Use a Customer Connection ID, credit card, or purchase order to place your order. PeopleSoft Internet site: http://www.peoplesoft.com/.
Telephone	Contact Consolidated Publishing Incorporated (CPI) at **800 888 3559.**
Email	Send email to CPI at **callcenter@conpub.com**.

PeopleBooks Standard Field Definitions

Throughout our product documentation, you will encounter fields and buttons that are used on many application pages or panels. This section lists the most common fields and buttons and provides standard definitions.

Field	*Definition*
As of Date	The last date for which a report or process includes data.
Business Unit	An identification code 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	Freeflow text up to 30 characters.
Effective Date	Date on which a table row becomes effective; the date that an action begins. For example, if you want 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. For more information about effective dates, see **Understanding Effective Dates in *Using PeopleSoft Applications.***

Field	*Definition*
EmplID (employee ID)	Unique identification code for an individual associated with your organization.
Language or **Language Code**	The language in which you want the field labels and report headings of your reports to print. The field values appear as you enter them. Language also refers to the language spoken by an employee, applicant, or non-employee.
Process Frequency group box	Designates the appropriate frequency in the **Process Frequency** group box: **Once** executes the request the next time the batch process runs. After the batch process runs, the process frequency is automatically set to **Don't Run**. **Always** executes the request every time the batch process runs. **Don't Run** ignores the request when the batch process runs.
Report ID	The report identifier.
Report Manager	This button takes you to 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	This button takes you to the Process List page, where you can view the status of submitted process requests.
Run	This button takes you to the Process Scheduler request page, where you can specify the location where a process or job runs and the process output format.
	For more information about the Report List page, the Process List page, and the Process Scheduler, see Process Scheduler Basics in the PeopleTools documentation.
Request ID	A request identification that represents a set of selection criteria for a report or process.
User ID	The system identifier for the individual who generates a transaction.
SetID	An identification code that represents a set of control table information or TableSets. A TableSet is a group of tables (records) necessary to define your company's structure and processing options.
Short Description	Freeflow text up to 15 characters.

Typographical Conventions and Visual Cues

We use a number of standard conventions and visual cues in our online documentation.

The following list contains our typographical conventions and visual cues:

`(monospace font)`	Indicates a PeopleCode program or other program example.
Bold	Indicates field names and other page elements, such as buttons and group box labels, when these elements are documented below the page on which they appear. When we refer to these elements elsewhere in the documentation, we set them in Normal style (not in bold). We also use boldface when we refer to navigational paths, menu names, or process actions (such as **Save** and **Run**).
Italics	Indicates a PeopleSoft or other book-length publication. We also use italics for *emphasis* and to indicate specific field values. When we cite a field value under the page on which it appears, we use this style: ***field value.*** 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.
Cross-references	The phrase **For more information** indicates where you can find additional documentation on the topic at hand. We include the navigational path to the referenced topic, separated by colons (:). Capitalized titles in *italics* indicate the title of a PeopleBook; capitalized titles in normal font refer to sections and specific topics within the PeopleBook. Here's an example: **For more information,** see **Documentation on CD-ROM** in *About These PeopleBooks:* Additional Resources.

Note. Text in this bar indicates information that you should pay particular attention to as you work with your PeopleSoft system. If the note is preceded by **Important!,** the note is crucial and includes information that concerns what you need to do for the system to function properly.

Text in this bar indicates cross-references to related or additional information.

Warning! Text within this bar indicates a crucial configuration consideration. Pay very close attention to these warning messages.

Page and Panel Introductory Table

In the documentation, each page or panel description in the application includes an introductory table with pertinent information about the page. Not all of the information will be available for all pages or panels.

Usage	Describes how you would use the page or process.
Object Name	Gives the system name of the panel or process as specified in the PeopleTools Application Designer. For example, the Object Name of the Detail Calendar panel is DETAIL_CALENDAR1.
Navigation	Provides the path for accessing the page or process.
Prerequisites	Specifies which objects must have been defined before you use the page or process.
Access Requirements	Specifies the keys and other information necessary to access the page. For example, **SetID** and **Calendar ID** are required to open the Detail Calendar page.

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 our documentation, 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 comments by email to the authors of the PeopleSoft documentation at:

DOC@PEOPLESOFT.COM

While we cannot guarantee to answer every email message, we will pay careful attention to your comments and suggestions. We are always improving our product communications for you.

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 A-C
- PeopleCode built-in functions and language constructs D-G
- PeopleCode built-in functions and language constructs H-R
- PeopleCode built-in functions and language constructs S-Z

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:

`monospace font`	Indicates a PeopleCode program or other example
Bold	In PeopleCode syntax, boldface items indicate function names, method names, language constructs, and PeopleCode reserved words that must be included literally in the function call.
Italics	In PeopleCode syntax, italic items 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.
[ ]	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

GetSession
%Session

Application Classes

CollectGarbage

Application Engine

CallAppEngine
CommitWork
GetAESection

Arrays

CreateArray
CreateArrayAny
CreateArrayRept
Split

Attachment

AddAttachment
CopyAttachments
DeleteAttachment
GetAttachment
PutAttachment
ViewAttachment

See Also

Files, Images

Bulk

BulkDeleteField

BulkInsertField
BulkModifyPageFieldOrder

Business Interlink

GetBiDoc
GetInterlink
%MaxInterlinkSize

Chart

CreateObject
GetChart
GetChartURL

ChartField

RenameDBField
RenamePage
RenameRecord
SetDBFieldAuxFlag
SetDBFieldCharDefn
SetDBFieldFormat
SetDBFieldFormatLength
SetDBFieldLabel
SetDBFieldLength
SetDBFieldNotUsed
SetPageFieldPageFieldName
SetRecFieldEditTable
SetRecFieldKey

Component Buffer

ActiveRowCount
AddKeyListItem
CompareLikeFields
ClearKeyList
ComponentChanged
CopyFields
CopyRow
CurrentLevelNumber
CurrentRowNumber
DeleteRecord
DeleteRow
DiscardRow
ExpandBindVar
ExpandEnvVar
ExpandSqlBinds
FetchValue
FieldChanged
GetNextNumber
GetNextNumberWithGaps
GetRelField

GetSetId
InsertRow
PriorValue
RecordChanged
RecordDeleted
RecordNew
RowFlush
SetChannelStatus
SetDefault
SetDefaultAll
SetTempTableInstance
StopFetching
TotalRowCount
TreeDetailInNode
UpdateSysVersion
UpdateValue
%Component
%Menu
%Mode
%OperatorClass
%Table
%TruncateTable

See Also

Data Buffer Access

Component Interface

GetMethodNames
GetSession
StartWork
%CompIntfcName

Conversion

Char
Code
Codeb
ConvertChar
NumberToString
String
Value

Currency and Financial

Amortize
ConvertCurrency
RoundCurrency
SinglePaymentPV
UniformSeriesPV
%Currency

Current Date and Time

%CurrentDateIn
%CurrentDateOut
%CurrentDateTimeIn
%CurrentDateTimeOut
%CurrentTimeIn
%CurrentTimeOut

See Also

Date and Time, SQL Date and Time

Custom Display Formats

GetStoredFormat
SetDisplayFormat

Database and Platform

%DbName
%DbType

Data Buffer Access

CreateRecord
CreateRowset
FlushBulkInserts
GetField
GetLevel0
GetRecord
GetRow
GetRowset

See Also

Component Buffer

Date and Time

AddToDate
AddToDateTime
AddToTime
ConvertDatetimeToBase
Date
Date3
DatePart
DateTime6
DateTimeToHTTP
DateTimeToLocalizedString
DateTimeToTimeZone
DateTimeValue

DateValue
Day
Days
Days360
Days365
FormatDateTime
GetCalendarDate
Hour
IsDaylightSavings
Minute
Month
Second
Time
Time3
TimePart
TimeToTimeZone
TimeValue
TimeZoneOffset
Weekday
Year
%AsOfDate
%ClientDate
%ClientTimeZone
%Date
%DateAdd
%DateDiff
%DateTime
%DateTimeDiff
%DateTimeIn
%DateTimeOut
%DTTM
%PermissionLists
%ServerTimeZone
%Time
%TextIn

See Also

Current Date and Time, SQL Date and Time

Defaults, Setting

SetDefault
SetDefaultAll
SetDefaultNext
SetDefaultNextRel
SetDefaultPrior
SetDefaultPriorRel

Double-Byte Characters

CharType
ContainsCharType
ContainsOnlyCharType
ConvertChar

DBCSTrim

Effective Date and Effective Sequence

CurrEffDt
CurrEffRowNum
CurrEffSeq
NextEffDt
NextRelEffDt
PriorEffDt
PriorRelEffDt
SetDefaultNext
SetDefaultNextRel
SetDefaultPrior
SetDefaultPriorRel
%EffDtCheck

Email

AddEmailAddress
ChangeEmailAddress
DeleteEmailAddress
MarkPrimaryEmailAddress

Environment

ExpandEnvVar
GetCwd
GetEnv

Exceptions

CreateException
Throw
Try

Executable Files, Running

Exec
WinExec

See Also

Object

Files

CreateDirectory
FileExists
FindFiles

GetFile
RemoveDirectory

See Also

Images, Attachment

Financial

AccruableDays
AccrualFactor
BlackScholesCall
BlackScholesPut
BootstrapYTMs
ConvertRate
CubicSpline
HermiteCubic
HistVolatility
LinearInterp

Grids

GetGrid

Images

DeleteImage
GetImageExtents
InsertImage

See Also

Attachment, Files

Import Manager

%ExternalAuthInfo

Integration Broker

EncryptNodePswd
FindCodeSetValues
NodeDelete
NodeRename
NodeSaveAs
PingNode

Transform

Internet

CreateSOAPDoc
EncodeURL
EncodeURLForQueryString
EscapeHTML
EscapeJavascriptString
EscapeWML
GenerateActGuideContentUrl
GenerateActGuidePortalUrl
GenerateActGuideRelativeUrl
GenerateComponentContentRelURL
GenerateComponentContentURL
GenerateComponentPortalRelURL
GenerateComponentPortalURL
GenerateComponentRelativeURL
GenerateExternalPortalURL
GenerateExternalRelativeURL
GenerateHomepagePortalURL
GenerateHomepageRelativeURL
GenerateQueryContentURL
GenerateQueryPortalURL
GenerateQueryRelativeURL
GenerateScriptContentRelURL
GenerateScriptContentURL
GenerateScriptPortalRelURL
GenerateScriptPortalURL
GenerateScriptRelativeURL
GenerateTree
GenerateWorklistPortalURL
GenerateWorklistRelativeURL
GetChartURL
GetHTMLText
GetMethodNames
GetURL
Unencode
ViewURL
%EmailAddress
%Request
%Response
%RunningInPortal

Java

CreateJavaArray
CreateJavaObject
GetJavaClass

Language Preference and Locale

SetLanguage
%IsMultiLanguageEnabled
%Language
%Language_Base
%Market

Language Constructs

Break
Component
Declare Function
Evaluate
Exit
For
Function
Global
If
Local
Repeat
Return
Throw
Try
While

Logical (Tests for Blank Values)

All
AllOrNone
None
OnlyOne
OnlyOneOrNone

Mail

SendMail
%EmailAddress

Math

Abs
Acos
Asin
Atan
Cos
Cot
Degrees
Exp
Fact
GetNextNumber
GetNextNumberWithGaps
Idiv
Int
Integer
Ln
Log10
Max
Min
Mod
Product
Radians

Rand
Round
Sign
Sin
Sqrt
Tan
Truncate
%Abs
%DecDiv
%DecMult
%Round
%Truncate

Menu Appearance

CheckMenuItem
DisableMenuItem
EnableMenuItem
HideMenuItem
UnCheckMenuItem

Message Catalog and Message Display

Error
MessageBox
MsgGet
MsgGetExplainText
MsgGetText
Prompt
Warning
WinMessage

Messages

AddSystemPauseTimes
CreateMessage
DeleteSystemPauseTimes
GetMessage
GetSyncLogData
IsMessageActive
ReturnToServer
PingNode
SetChannelStatus
%MaxMessageSize

See Also

XML

Modal Transfers

DoModalComponent
EndModalComponent

IsModalComponent

See Also

Secondary Pages, Transfers

Object-Based

CreateArray
CreateArrayRept
CreateException
CreateJavaArray
CreateJavaObject
CreateMessage
CreateObject
CreateObjectArray
CreateProcessRequest
CreateRecord
CreateRowset
CreateSOAPDoc
CreateSQL
CreateXmlDoc
DeleteSQL
FetchSQL
GetAESection
GetChart
GetChartURL
GetCwd
GetField
GetFile
GetGrid
GetHTMLText
GetInterlink
GetLevel0
GetJavaClass
GetMessage
GetMessageXmlDoc
GetMessageInstance
GetPubContractInstance
GetRecord
GetRow
GetRowset
GetSession
GetSQL
GetSubContractInstance
ObjectDoMethod
ObjectDoMethodArray
ObjectGetProperty
ObjectSetProperty
ReturnToServer
Split
StoreSQL

Object

CreateObject
CreateObjectArray
ObjectDoMethod
ObjectDoMethodArray
ObjectGetProperty
ObjectSetProperty

Page

GetPage

Page Control Appearance

AddDropDownItem
ClearDropDownList
GetImageExtents
Gray
Hide
HideRow
HideScroll
IsHidden
SetCursorPos
SetLabel
Ungray
Unhide
UnhideRow
UnhideScroll

Personalizations

GetUserOption
SetUserOption

Process Scheduler

CreateProcessRequest
GetNextProcessInstance

SetupScheduleDefnItem

Remote Call

DoSaveNow
RemoteCall

Saving and Canceling

DoCancel
DoSave
DoSaveNow
WinEscape

Scroll Select

RowFlush
RowScrollSelect
RowScrollSelectNew
ScrollFlush
ScrollSelect
ScrollSelectNew
SortScroll

Search Dialog

ClearSearchDefault
ClearSearchEdit
IsSearchDialog
SetSearchDefault
SetSearchDialogBehavior
SetSearchEdit
%Mode

Secondary Pages

DoModal
EndModal
IsModal

See Also

Modal Transfers, Transfers

SQL

CreateSQL
DeleteSQL
ExpandBindVar
ExpandSqlBinds
FetchSQL
FlushBulkInserts
GetSQL
SQLExec
StoreSQL
%FirstRows
%InsertSelect
%InsertValues
%Join

%KeyEqual
%KeyEqualNoEffDt
%Like
%LikeExact
%List
%ListBind
%ListEqual
%NoUppercase

%OldKeyEqual
%SQL
%SignonUserId
%Table
%UpdatePairs

See Also

Scroll Select, Data Buffer Access

SQL Date and Time

%DateAdd
%DateDiff
%DateIn
%DateOut
%DateTimeDiff
%DateTimeIn
%DateTimeOut
%DTTM
%TimeAdd
%TextIn
%TimeIn
%TimeOut

See Also

Current Date and Time, Date and Time

SQL Shortcuts

%Delete
%Insert
%SelectAll
%SelectDistinct
%SelectByKey
%SelectByKeyEffDt
%Update

String

Clean
Code
Codeb

DBCSTrim
DBPatternMatch
Exact
ExpandBindVar
ExpandEnvVar
Find
Findb
GetHTMLText
IsAlpha
IsAlphaNumeric
IsDigits
Left
Len
Lenb
Lower
LTrim
NumberToString
Proper
Replace
Rept
Right
RTrim
String
Substitute
Substring
Substringb
Upper
%Abs%Substring
%TrimSubstr
%Upper

Subrecords

%SUBREC

TimeZone

ConvertDatetimeToBase
ConvertTimeToBase
DateTimeToTimeZone
FormatDateTime
IsDaylightSavings
TimeToTimeZone
TimeZoneOffset
%ClientTimeZone
%ServerTimeZone

Trace Control

SetTracePC
SetTraceSQL

Transfers

AddKeyListItem
ClearKeyList
SetNextPage
Transfer
TransferNode
TransferPage
TransferPortal

See Also

Modal Transfers, Secondary Pages

Type Checking

IsAlpha
IsAlphaNumeric
IsDate
IsDateTime
IsDigits
IsNumber
IsTime
Max
Min
NumberToString

User Information

%EmployeeId
%EmailAddress
%UserDescription
%UserId

User Security

AllowEmplIdChg
Decrypt
Encrypt
ExecuteRolePeopleCode
ExecuteRoleQuery
ExecuteRoleWorkflowQuery
Hash
IsMenuItemAuthorized
IsUserInPermissionList
IsUserInRole
ReturnToServer
SetAuthenticationResult
SetPasswordExpired
SwitchUser
%AuthenticationToken
%EmailAddress
%ExternalAuthInfo

%NavigatorHomePermissionList
%PasswordExpired
%PermissionLists
%PrimaryPermissionList
%ProcessProfilePermissionList
%PSAuthResult
%ResultDocument
%Roles
%RowSecurityPermissionList
%SignonUserId
%SignOnUserPswd
%UserId

Validation

Error
IsMenuItemAuthorized
RevalidatePassword
SetCursorPos
SetReEdit
Warning

Workflow

GenerateActGuideContentUrl
GenerateActGuidePortalUrl
GenerateActGuideRelativeUrl
GetWLFieldValue
MarkWLItemWorked
TriggerBusinessEvent
%BPName
%WLInstanceID
%WLName

XML

CancelPubHeaderXmlDoc
CancelPubXmlDoc
CancelSubXmlDoc
CreateSOAPDoc
CreateXmlDoc
GetMessageXmlDoc
GetNRXmlDoc
GetPubHeaderXmlDoc
GetPubXmlDoc
GetSubXmlDoc
GetSyncLogData
PublishXmlDoc
ReSubmitPubHeaderXmlDoc
ReSubmitPubXmlDoc
ReSubmitSubXmlDoc
ReValidateNRXmlDoc
SyncRequestXmlDoc
Transform

UpdateXmlDoc

PeopleCode Built-in Functions and Language Constructs A-C

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

Sign, %Abs

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

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:

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

Returns

An integer representing a number of days.

See Also

AccrualFactor

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

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:

Value	*Constant*	*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
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

AccruableDays

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

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

Asin, Atan, Cos, Cot, Degrees, Radians, Sin, Tan

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.

See Also

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," Specifying Data With References Using Scroll Path Syntax and Dot Notation

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. See also *PeopleTools PeopleBook: PeopleCode Developer's Guide*, "Data Buffer Access".

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.

Returns

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

Parameters

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

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

CurrentRowNumber, TotalRowCount

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," Specifying Data With References Using Scroll Path Syntax and Dot Notation

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 shouldn't 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.

See Also

GetAttachment, PutAttachment

"Rowset Class", Select, SelectNew

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-in Functions," Think-Time Functions

Parameters

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.

Returns

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

Number 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.
3	%Attachment_FileTransferFailed	File transfer didn't succeed.

Number Value	Constant Value	Description
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.

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

DeleteAttachment, GetAttachment, PutAttachment, ViewAttachment

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-in Functions," Understanding File Attachment Architecture

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

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

ClearDropDownList

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

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

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

ChangeEmailAddress, DeleteEmailAddress, MarkPrimaryEmailAddress

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.

Returns

Returns a Boolean value indicating whether it completed successfully.

Parameters

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

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

ClearKeyList, TransferPage, Transfer

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

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

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

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

DeleteSystemPauseTimes

PeopleTools PeopleBook: Integration Tools, "Chapter Name," 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 couldn't 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.

Returns

Returns a Date value equal to the original date plus the number of years, months, and days passed to the function.

Parameters

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.

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

DateValue, Day, Days, Days360, Days365, Month, Weekday

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.

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.

Parameters

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.*

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

AddToTime, DateValue, DateTimeValue, TimeValue

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.)

Returns

A Time value equal to time increased by the number of hours, minutes, and seconds passed to the function.

Parameters

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*.

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

AddToDateTime, DateValue, DateTimeValue, TimeValue

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.

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.

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.

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

AllOrNone, None, OnlyOne, OnlyOneOrNone, SetDefault, SetDefaultAll

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.

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.

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.

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

All, None, OnlyOne, OnlyOneOrNone

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.

Returns

Optionally returns a Boolean value: True if the function executed successfully, False otherwise.

Parameters

is_allowed	A Boolean value indicating whether the user is permitted to change the user's own data.

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*.

Returns

None.

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.

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.

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

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

Acos, Atan, Cos, Cot, Degrees, Radians, Sin, Tan

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

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

Acos, Asin, Cos, Cot, Degrees, Radians, Sin, Tan

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

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

BlackScholesPut

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

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

BlackScholesCall

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

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

"Array Class"

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

Evaluate, Exit, For, While

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.

Information about this operation is stored in a log field. The log file is placed in the following directory:

```
PS_HOME/APPSERV/Domain_Name/LOGS/BulkOps.txt
```

where *Domain_Name* is the name of the application server domain.

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 fieldname you deleted.

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 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

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:

Constant 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

BulkInsertField, BulkModifyPageFieldOrder

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Debugging Your Application," Find In . . .

PeopleTools PeopleBook: Peoplesoft Application Designer, "Appendix F: Performing Bulk Operations"

PeopleTools PeopleBook: 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.

Information about this operation is stored in a log field. The log file is placed in the following directory:

```
PS_HOME/APPSERV/Domain_Name/LOGS/BulkOps.txt
```

where *Domain_Name* is the name of the application server domain.

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

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:

Constant 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

BulkModifyPageFieldOrder, BulkDeleteField

PeopleTools PeopleBook: Peoplesoft Application Designer, "Appendix F: Performing Bulk Operations"

PeopleTools PeopleBook: 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.

Information about this operation is stored in a log field. The log file is placed in the following directory:

```
PS_HOME/APPSERV/Domain_Name/LOGS/BulkOps.txt
```

where *Domain_Name* is the name of the application server domain.

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

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:

Constant 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

BulkInsertField, BulkDeleteField

PeopleTools PeopleBook: Peoplesoft Application Designer, "Appendix F: Performing Bulk Operations"

PeopleTools PeopleBook: Peoplesoft Application Designer, "Building SQL Tables and Views"

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.

See Also

PeopleTools PeopleBook: Peoplesoft Application Engine, "Using Meta-SQL and PeopleCode," Using PeopleCode in Programs

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.

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

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

DoSaveNow

"AESection Class"

"Rowset Class", Refresh

PeopleTools PeopleBook: Peoplesoft Application Engine, "Using Meta-SQL and PeopleCode," Using PeopleCode in Programs

PeopleTools PeopleBook: 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

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

ReSubmitPubHeaderXmlDoc

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

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

ReSubmitPubXmlDoc

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

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

ReSubmitSubXmlDoc

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

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

Address Specify the email address that you want to add as a string.

Returns

None.

See Also

AddEmailAddress, DeleteEmailAddress, MarkPrimaryEmailAddress

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

Code, String

"Meta-SQL", %Substring

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 occurd 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

PeopleTools PeopleBook: PeopleSoft Global Technology, "Character Sets and Language Input/Output"

Returns

CharType returns one of the following Number values. You can check for the constant values instead of the numeric values if you prefer:

Return Value	Constant	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.

Parameters

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:

Character Code	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

Character Code	Constant	Character Set
10	%CharType_JapanesePunctuation	Japanese punctuation

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

ContainsCharType, ContainsOnlyCharType, ConvertChar

PeopleTools PeopleBook: PeopleSoft 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

Char, String

"Meta-SQL", %Substring

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 overriden 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

AddDropDownItem

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

AddKeyListItem

ClearSearchDefault

Syntax

ClearSearchDefault([*recordname*.]*fieldname*)

Description

ClearSearchDefault disables default processing for the specified field, reversing the effects of a previous call to **SetSearchDefault**. 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.

Note. This function remains for backward compatibility only. Use the SearchDefault Field class property instead. See *PeopleTools PeopleBook: PeopleCode Developer's Guide,* "Data Buffer Access".

Parameters

[*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

ClearSearchEdit, SetSearchDefault, SetSearchEdit, SetSearchDialogBehavior

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Understanding 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 *PeopleTools PeopleBook: PeopleCode Developer's Guide,* "Data Buffer Access."

Parameters

[*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

SetSearchEdit, SetSearchDefault, ClearSearchDefault, SetSearchDialogBehavior

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Understanding 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

Char, String, Codeb

"Meta-SQL," %Substring

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

Char, String, Code

"Meta-SQL," %Substring

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

"Application Classes"

CommitWork

Syntax

```
CommitWork()
```

Description

The **CommitWork** function commits pending changes (inserts, updates, and deletes) to the database.

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 via CallAppEngine, the **CommitWork** will be 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.

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 Also

PeopleTools PeopleBook: 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 Also

PeopleTools PeopleBook: Peoplesoft Application Engine, " Managing Application Engine Programs," Restarting Application Engine Programs

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

PeopleTools PeopleBook: 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.

See Also

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," Specifying Data With References Using Scroll Path Syntax and Dot Notation

Description

CompareLikeFields compares fields in a row on a specified source record to similarly named fields on a specified target record. If all of the like-named fields have the same data value, **CompareLikeFields** returns True; otherwise it returns False.

Note. This function remains for backward compatibility only. Use the CompareFields record class method instead. See *PeopleTools PeopleBook: PeopleCode Developer's Guide*, "Data Buffer Access."

Returns

Returns a Boolean value indicating whether all of the like-named fields in the two records have the same data value.

Parameters

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.

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

CopyFields

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," Specifying Data With References Using Scroll Path Syntax and Dot Notation

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

data_type	Specify a PeopleCode data type.
&*var_name*	A legal variable name.

Example

```
Component string &PG_FIRST;
```

See Also

Local, Global

PeopleTools PeopleBook: 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.

Returns

ContainsCharType returns one of the following Number values. You can check for the constant instead of the numeric value if you prefer:

Return Value	Constant	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.

Parameters

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 or a Constant:

Character Code	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

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

ContainsCharType, ContainsOnlyCharType, ConvertChar

PeopleTools PeopleBook: PeopleSoft 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.

Returns

ContainsOnlyCharType returns one of the following Number values. You can check for the constant instead of the numeric value, if you prefer:

Return Value	*Constant*	*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***.

Return Value	Constant	Description
-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.

Parameters

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 or a constant:

Character Code	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

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

CharType, ContainsCharType, ConvertChar

PeopleTools PeopleBook: PeopleSoft 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).

Returns

Returns either a Number or a constant with one of the following values, depending on what you're checking for:

Value	*Constant*	*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.

Parameters

Source_str	String to be converted.
Source_str_category	Language category of input string. You can specify either a number or a constant.

Category	Constant	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)

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 or a constant:

Character Code	Constant	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.

Character Code	Constant	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
	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

Source	Target	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

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

CharType, ContainsCharType, ContainsOnlyCharType

PeopleTools PeopleBook: PeopleSoft 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 PeopleBook: Peoplesoft Global Technologies, "Controlling Currency Display Format"

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.

Parameters

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.

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

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

ConvertTimeToBase, FormatDateTime, IsDaylightSavings, DateTimeToTimeZone, TimeToTimeZone, TimeZoneOffset

PeopleTools PeopleBook: Data Management, "PeopleTools Utilities"

ConvertRate

Syntax

```
ConvertRate(Rate, In_Frequency, Out_Frequency)
```

Description

The **ConvertRate** function converts a rate between various compounding frequencies.

Parameters

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

Value	Description
2	monthly compounding
3	yearly compounding

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

RoundCurrency

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** will do 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

texttime	Specify a time value represented as text (e.g., "3:00 PM")
timezone | **Local** | **Base**	Specify a value for converting *textdatetime*. The valid 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

ConvertDatetimeToBase, FormatDateTime, IsDaylightSavings, DateTimeToTimeZone, TimeToTimeZone, TimeZoneOffset

PeopleTools PeopleBook: Data Management, "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

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:

Number	Constant	Description
0	%Attachment_Success	File was transferred successfully.
1	%Attachment_Failed	File was not successfully transferred.
3	%Attachment_FileTransferFailed	File transfer didn't succeed.

Number	Constant	Description
7	%Attachment_DestSystNotFound	Cannot locate destination system.
8	%Attachment_DestSystFailedLogin	Unable to login to destination system.

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

AddAttachment, DeleteAttachment, GetAttachment, PutAttachment, ViewAttachment

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-in Functions," Understanding File Attachment Architecture

PeopleTools PeopleBook: Data 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.

See Also

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," Specifying Data With References Using Scroll Path Syntax and Dot Notation

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 *PeopleTools PeopleBook: PeopleCode Developer's Guide*, "Data Buffer Access". Also see CopyTo Row class method, and the CopyTo Rowset class method.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Parameters

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.

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

CompareLikeFields

PeopleTools PeopleBook: 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. *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.

Note. This function remains for backward compatibility only. Use the CopyTo Row class method instead. See *PeopleTools PeopleBook: PeopleCode Developer's Guide*, "Data Buffer Access". Also see CopyTo Rowset class method, as well as CopyFieldsTo and CopyChangedFieldsTo Record class methods.

Parameters

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

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

Acos, Asin, Atan, Cot, Degrees, Radians, Sin, Tan

Cot

Syntax

Cot(*angle*)

Description

Cot calculates the cotangent of the given angle (adjacent / opposite, that is, the reciprocal of the tangent).

Parameters

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

Acos, Asin, Atan, Cos, Degrees, Radians, Sin, Tan

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

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

CreateArrayRept, Split

"Array Class," Using Flattening and Promotion

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

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

CreateArrayRept, CreateArray, Split

"Array Class," Using Flattening and Promotion

"SQL Class"

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

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

CreateArray, Split

"Array Class," Clone

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]
3. PS_SERVDIR\Files[\relative_path]
4. 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

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

FileExists, FindFiles, GetFile, GetAttachment, PutAttachment, RemoveDirectory

"File Class"

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

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

"Exception Class"

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

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

CreateJavaObject, GetJavaClass

"Java Class"

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

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

CreateJavaArray, GetJavaClass

"Java Class"

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

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

GetMessage, GetPubContractInstance, GetSubContractInstance

"Message Class"

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

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

ObjectDoMethod, ObjectGetProperty, ObjectSetProperty, CreateObjectArray, ObjectDoMethodArray

"Internet Script Classes (iScript)"

PeopleTools PeopleBook: 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

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

CreateObject, ObjectDoMethod, ObjectGetProperty, ObjectSetProperty, ObjectDoMethodArray

"Array Class"

PeopleTools PeopleBook: 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

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

"ProcessRequest Class", Schedule

PeopleTools PeopleBook: Peoplesoft Process Scheduler, "Understanding PeopleSoft Process Scheduler"

CreateRecord

Syntax

```
CreateRecord(RECORD.recname)
```

Description

CreateRecord creates a **freestanding** 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**.

Returns

This function returns a record object that references a new record buffer and set of fields.

Parameters

RECORD.*recname*	Specify a record definition that already exists.

Examples

```
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

GetRecord, GetField

"Record Class"

PeopleTools PeopleBook: 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

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);
```

See Also

GetRowset, GetLevel0, GetRecord, GetField

PeopleTools PeopleBook: 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

"SOAPDoc Class"

"XmlDoc Class"

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

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

DeleteSQL, FetchSQL, GetSQL, SQLExec, StoreSQL

"SQL Class," Open

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.

Parameters

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

"XmlDoc Class"

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

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:

Value	Constant	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
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

Value	Constant	Description
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.

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

HermiteCubic, LinearInterp

CurrEffDt

Syntax

```
CurrEffDt([level_num])
```

Description

CurrEffDt returns the effective date of the specified scroll level as a Date value. If no level is specified, **CurrEffDt**, returns the effective date of the current scroll level.

Note. This function remains for backward compatibility only. Use the EffDt Rowset class property instead. See *PeopleTools PeopleBook: PeopleCode Developer's Guide,* "Data Buffer Access". Also see GetCurrEffRow Rowset class method and EffSeq Rowset class property.

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

CurrEffRowNum, CurrEffSeq, CurrentLevelNumber

CurrEffRowNum

Syntax

```
CurrEffRowNum([level_num])
```

Description

CurrEffRowNum returns the effective row number of the selected scroll level. If no level is specified, it returns the effective row number of the current level.

Note. This function remains for backward compatibility only. Use the RowNumber Row class property, in combination with the GetCurrEffRow Rowset method instead. See also *PeopleTools PeopleBook: PeopleCode Developer's Guide,* "Data Buffer Access".

Example

```
&ROW = CurrEffRowNum(1);
```

See Also

CurrEffSeq , CurrentLevelNumber , CurrEffRowNum

CurrEffSeq

Syntax

CurrEffSeq([*level_num*])

Description

CurrEffSeq determines the effective sequence of a specific scroll area. If no level is specified, **CurrEffSeq** returns the effective sequence of the current scroll level.

Note. This function remains for backward compatibility only. Use the EffSeq rowset class property instead. See *PeopleTools PeopleBook: PeopleCode Developer's Guide,* "Data Buffer Access". Also see GetCurrEffRow Rowset method and DeleteEnabled Rowset property.

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

CurrEffDt, CurrentLevelNumber**,** CurrEffRowNum

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

CurrentRowNumber, FetchValue

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. See *PeopleTools PeopleBook: PeopleCode Developer's Guide,* "Data Buffer Access". Also see GetRow function.

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.

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.

Parameters

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.

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

ActiveRowCount, CurrentLevelNumber, FetchValue

PeopleCode Built-in Functions and Language Constructs D-G

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 will be 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

Date3, DateValue, Day, Days360, Days365, Month, Weekday, Year

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.

Returns

Returns a Date value equal to the date specified in the function parameters.

Parameters

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.

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

Date, DateValue, Day, Days360, Days365

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.

Returns

Returns a DateTime value based on the integers provided.

Parameters

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.

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 DateTimeToLocalizedString 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

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

AddToDateTime, DateTimeToLocalizedString, FormatDateTime

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

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. See Using the Pattern Parameter.

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
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

Symbol	Definition	Type	Example
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.
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

FormatDateTime, DateTimeToHTTP

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 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

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

ConvertDatetimeToBase, ConvertTimeToBase, FormatDateTime, IsDaylightSavings, TimeToTimeZone, TimeZoneOffset

PeopleTools PeopleBook: Data Administration, "PeopleTools Utilities"

DateTimeValue

Syntax

```
DateTimeValue(string)
```

Description

Use **DateTimeValue** to derive a DateTime value from a string representing a date and time. The string has the form:

```
MM/DD/YY[YY] hh:mm:ss.ssssss [{AM|PM}]
```

or

```
MM.DD.YY[YY] hh:mm:ss.ssssss [{AM|PM}]
```

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

Date, Date3, DateValue, Day, Days360, Days365, Month, Weekday, Year

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

Date, Date3, DateTimeValue, Day, Days360, Days365, Month, Weekday, Year

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

Date, Date3, DateValue, Days, Days360, Days365, Month, Weekday, Year

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

DateValue, Days360, Days365, Month, Weekday, Year

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

Date, Date3, DateValue, Day, Days, Days365, Month, Weekday, Year

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 = Days365(TERM_START_DT, PMT_DT);
```

See Also

Date, Date3, DateValue, Day, Days360, Month, Weekday, Year

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

Substringb

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

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

Find, Findb

Declare Function

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   ([ext_param1 [, ext_param2] …)
ext_paramn  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 Also

PeopleTools PeopleBook: 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, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.

See Think-Time Functions.

PeopleCode Function Parameters

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.

External Function Parameters

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.

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

Function

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

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

Encrypt, Hash, Security

Degrees

Syntax

```
Degrees(angle)
```

Description

Degrees converts the given angle from radian measurement to degree measurement.

Parameters

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

Acos, Asin, Atan, Cos, Cot, Radians, Sin, Tan

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

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:

Number	Constant	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.

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

AddAttachment, GetAttachment, PutAttachment, ViewAttachment

PeopleTools PeopleBook: Peoplesoft 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

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

AddEmailAddress, ChangeEmailAddress, MarkPrimaryEmailAddress

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.

Restrictions for Use With Windows Client

This function is available only in the PeopleSoft Internet Architecture, and not with Windows client.

Parameters

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

InsertImage

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. **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.

Note. This function remains for backward compatibility only. Use the Delete Record class method instead. See *PeopleTools PeopleBook: PeopleCode Developer's Guide*, "Data Buffer Access."

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.

Returns

Optionally returns a Boolean value indicating whether the deletion was completed successfully.

Parameters

level_zero_recfield	A *recordname.fieldname* reference identifying any field on the level-zero area of the page.

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

DeleteRow

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.

See Also

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," Specifying Data With References Using Scroll Path Syntax and Dot Notation.

Description

Use **DeleteRow** to delete rows programmatically. A call to this function causes the RowDelete event sequence to fire, as if an user had manually deleted a row.

Note. This function remains for backward compatibility only. Use the DeleteRow Rowset class method instead. See *PeopleTools PeopleBook: PeopleCode Developer's Guide*, "Data Buffer Access."

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).

Returns

Boolean (optional). **DeleteRow** returns a Boolean value indicating whether the deletion was completed successfully.

Parameters

scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number of the row to delete.

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

InsertRow

PeopleTools PeopleBook: 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

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:

- DEFAULT
- DB2ODBC
- 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

CreateSQL, FetchSQL, GetSQL, StoreSQL, CommitWork

"SQL Class"

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

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

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

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

AddSystemPauseTimes

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Using Integration Broker Monitor"

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

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

EnableMenuItem, HideMenuItem

DiscardRow

Syntax

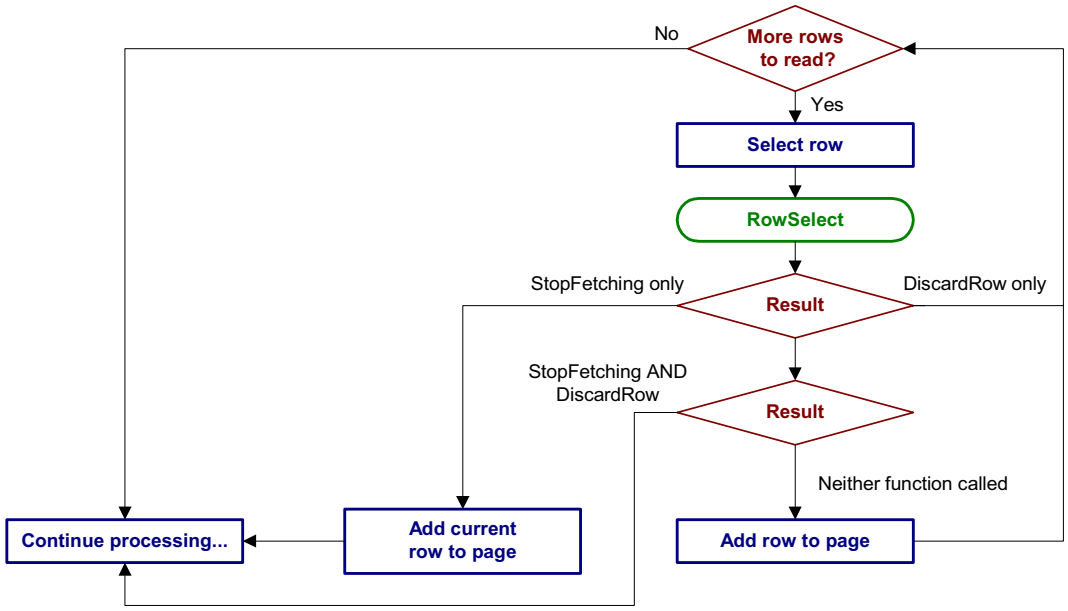
```
DiscardRow()
```

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

StopFetching

PeopleTools PeopleBook: Peoplesoft PeopleCode Developer's Guide, "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

DoSave, DoSaveNow, DoModal, EndModal, WinEscape

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.

See Also

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," Specifying Data With References Using Scroll Path Syntax and Dot Notation

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.

See Using Secondary Pages.

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.

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, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.

In addition, you can't use this function in SearchInit.

See Also

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.

Parameters

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

EndModal, IsModal, DoModalComponent

PeopleTools PeopleBook: 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

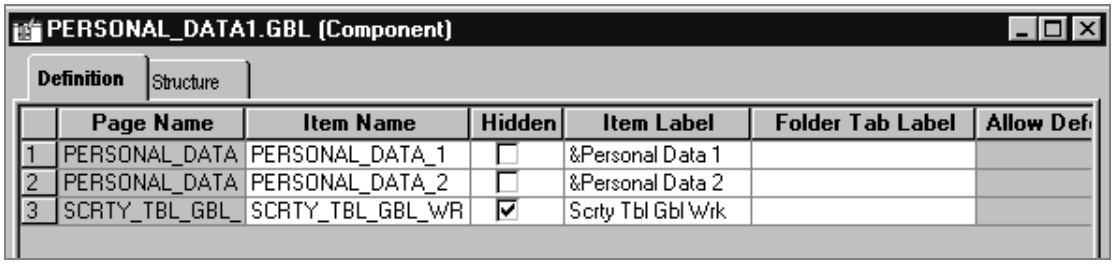
DoModalComponent launches a modal component. The modal component launches from within an originating component. After the modal component displays, the user can't proceed with changes to the originating component until either accepting or canceling the modal component.

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.

Parameters

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:

Value	Constant	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

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 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 **DoModalComponent** 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

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

IsModalComponent, Transfer, TransferPage, EndModalComponent

PeopleTools PeopleBook: 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.

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.

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

DoCancel, DoSaveNow, TransferPage, AddKeyListItem

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.

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

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

DisableMenuItem, HideMenuItem

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

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

EncodeURLForQueryString, Unencode

"Internet Script Classes (iScript)"

EncodeURLForQueryString

Syntax

EncodeURLForQueryString(*URLString*)

Description

The **EncodeURLForQueryString** function encodes *URLString* for use in a querystring 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

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 querystring 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

EncodeURL, Unencode, EscapeHTML, EscapeJavascriptString, EscapeWML
"Internet Script Classes (iScript)"

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

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

Decrypt, Hash, Security

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 decryption PeopleCode method.

Parameters

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

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Administering Basic Integrations," Configuring a Node

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** will 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.

Returns

None.

Parameters

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.

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

MessageBox, WinMessage

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.

See Also

PeopleTools PeopleBook: Peoplesoft PeopleCode Developer's Guide, "PeopleCode and PeopleSoft Internet Architecture"

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

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

DoModal, IsModal

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

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

DoModalComponent, EndModal, IsModal

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.

Returns

None.

Parameters

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.

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

MsgGet, MsgGetText, SetCursorPos, Warning, WinMessage

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

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

EscapeJavascriptString, EscapeWML

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

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

EscapeHTML, EscapeWML

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

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

EscapeHTML, EscapeJavascriptString

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 semi-colon 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 semi-colon 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

Len, String, %Substring

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.

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.

Parameters

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

Declare Function , RemoteCall, WinExec, ScrollSelect, ScrollSelectNew, RowScrollSelect, RowScrollSelectNew

"Rowset Class", Select

"Exception Class"

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using PeopleCode 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

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

ExecuteRoleQuery, ExecuteRoleWorkflowQuery, IsUserInPermissionList, IsUserInRole, %Roles, 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

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

ExecuteRolePeopleCode, ExecuteRoleWorkflowQuery, IsUserInPermissionList, IsUserInRole, %Roles, 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

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

ExecuteRolePeopleCode, ExecuteRoleQuery, IsUserInPermissionList, IsUserInRole, %Roles, 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.

Parameters

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

Break, Return,

PeopleTools PeopleBook: 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 (e^2):

```
&NUM = Exp(1);
&NUM = Exp(2);
```

See Also

Ln, Log10

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 Bind Variables in SQLExec and Inline Bind Variables in SQLExec for details about bind variables and inline bind variables.

Returns

Returns a String value equal to the input string with all bind variables expanded.

Parameters

str	A string containing one or more inline bind variables.

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

MessageBox, SQLExec

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

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

ExpandBindVar, GetEnv, GetCwd

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

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

SQLExec
"SQL Class"

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

Product

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

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 %DbName system variable.)

Values for *dbtype* are as follows. These values are not case-sensitive:

- DEFAULT
- DB2ODBC

- 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

CreateSQL, DeleteSQL, SQLExec, GetSQL, StoreSQL

"SQL Class"

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.

See Also

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," Specifying Data With References Using Scroll Path Syntax and Dot Notation

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. See PeopleTools PeopleBook: PeopleCode Developer's Guide, "Data Buffer Access". See also LongTranslateValue, ShortTranslateValue Field class properties.

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.

Returns

Returns the field value as an Any data type.

Parameters

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*.

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

ActiveRowCount, CopyRow, CurrentRowNumber, PriorValue, UpdateValue

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer"

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," Specifying Data With References Using Scroll Path Syntax and Dot Notation

FieldChanged

Syntax

The syntax of **FieldChanged** varies, depending on whether it uses a scroll path reference or a contextual reference to specify the field.

If it uses 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 it uses 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.

See Also

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer"

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," Specifying Data With References Using Scroll Path Syntax and Dot Notation

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. This is useful during SavePreChange or SavePostChange processing for checking whether to make related updates based on a change to a field.

Note. This function remains for backward compatibility only. Use the IsChanged Field class property instead. See *PeopleTools PeopleBook: PeopleCode Developer's Guide*, "Data Buffer Access".

Parameters

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

RecordChanged, RecordDeleted, RecordNew

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

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

FindFiles, GetFile

"File Class," Open

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.

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*.

Parameters

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*.

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

Exact, Len, DBPatternMatch

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*.

Returns

Returns a Number value indicating the starting position of *string* in *within_string*.

Parameters

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 byte of *within_string*.

Examples

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

Lenb

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

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 2 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

"Array Class"

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Applying Transformation, Translation and Filtering"

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

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

FileExists, GetFile

"File Class"

"Array Class"

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

"SQL Class"

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 PeopleTools Utilities for details about setting the base time.

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

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

ConvertDatetimeToBase, ConvertTimeToBase, DateTimeToLocalizedString, IsDaylightSavings, DateTimeToTimeZone, TimeToTimeZone, TimeZoneOffset, DateTimeToHTTP

Function

Syntax

```
Function name [(paramlist)] [Returns data_type]
     [statements]
End-function

where

paramlist is       &param1 [As data_type] [, &param2
                   [As data_type]]...
 data_type is      Number, String, Date, Time, Datetime,
                   Boolean, Object, or Any
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.

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.

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 PeopleCode data types (Number, String, Date, DateTime, Time, Boolean, or Any). 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

Declare Function , Return

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

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

GenerateActGuidePortalUrl, GenerateActGuideRelativeUrl

PeopleTools PeopleBook: 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

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_SE
CURITY.CHANGE_PASSWORD.GBL
```

See Also

GenerateActGuideContentUrl, GenerateActGuideRelativeUrl

PeopleTools PeopleBook: Peoplesoft Workflow, "Designing Activity Guides"

GenerateActGuideRelativeUrl

Syntax

```
GenerateActGuideRelativeUrl(PORTAL.portalname
                            NODE.nodename,
                            MENUNAME.menuname,
                            Marketname,
                            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

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/1/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_P
ASSWORD.GBL
```

See Also

GenerateActGuideContentUrl, GenerateActGuidePortalUrl

PeopleTools PeopleBook: 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

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=comp
onent?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&&A
ction=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

GenerateComponentContentURL, GenerateComponentPortalRelURL, GenerateComponentPortalURL, GenerateComponentRelativeURL

"Internet Script Classes (iScript)"

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

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=componen
t?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?pag
e=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

GenerateComponentContentRelURL, GenerateComponentPortalRelURL, GenerateComponentPortalURL, GenerateComponentRelativeURL

"Internet Script Classes (iScript)"

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

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=comp
onent?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

GenerateComponentContentRelURL, GenerateComponentContentURL, GenerateComponentPortalURL, GenerateComponentRelativeURL

"Internet Script Classes (iScript)"

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

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/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

GenerateComponentContentRelURL, GenerateComponentContentURL, GenerateComponentPortalRelURL, GenerateComponentRelativeURL

"Internet Script Classes (iScript)"

GenerateComponentRelativeURL

```
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

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=comp
onent?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

GenerateComponentContentRelURL, GenerateComponentContentURL, GenerateComponentPortalRelURL, GenerateComponentPortalURL

"Internet Script Classes (iScript)"

GenerateExternalPortalURL

```
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

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

GenerateExternalRelativeURL

"Internet Script Classes (iScript)"

GenerateExternalRelativeURL

```
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

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

GenerateExternalRelativeURL

"Internet Script Classes (iScript)"

GenerateHomepagePortalURL

```
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

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:

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

GenerateHomepageRelativeURL

"PortalRegistry Classes"

"Internet Script Classes (iScript)"

GenerateHomepageRelativeURL

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

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

GenerateHomepagePortalURL

"PortalRegistry Classes"

"Internet Script Classes (iScript)"

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

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.QUER
YNAME
```

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=PUBL
IC.QUERYNAME
```

See Also

GenerateQueryPortalURL, GenerateQueryRelativeURL

"Internet Script Classes (iScript)"

"Query Classes"

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

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.QUER
YNAME
```

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=PUBL
IC.QUERYNAME
```

See Also

GenerateQueryContentURL, GenerateQueryRelativeURL

"Internet Script Classes (iScript)"

"Query Classes"

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

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

GenerateQueryContentURL, GenerateQueryPortalURL

"Internet Script Classes (iScript)"

"Query Classes"

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

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

GenerateScriptContentURL, GenerateScriptPortalRelURL, GenerateScriptPortalURL, GenerateScriptRelativeURL

"Internet Script Classes (iScript)"

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

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_na
me.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.SFASCRIP
TS.FieldFormula.IScript_SFAHOME
```

See Also

GenerateScriptContentRelURL, GenerateScriptPortalRelURL, GenerateScriptPortalURL, GenerateScriptRelativeURL

"Internet Script Classes (iScript)"

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

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.f
unction_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

GenerateScriptContentRelURL, GenerateScriptContentURL, GenerateScriptPortalURL, GenerateScriptRelativeURL

"Internet Script Classes (iScript)"

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

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_na
me.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.SFASCRIP
TS.FieldFormula.IScript_SFAHOME
```

See Also

GenerateScriptContentRelURL, GenerateScriptContentURL, GenerateScriptPortalRelURL, GenerateScriptRelativeURL

"Internet Script Classes (iScript)"

GenerateScriptRelativeURL

```
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

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.fu
nction_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_SFAH
OME
```

See Also

GenerateScriptContentRelURL, GenerateScriptContentURL, GenerateScriptPortalRelURL, GenerateScriptPortalURL

"Internet Script Classes (iScript)"

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.

See Building Your HTML Tree Page for details about creating the field used for the TreeEventField.

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

Using HTML Tree End-User Actions (Events)

Parameters

&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 PostBuild PeopleCode Example or FieldChange PeopleCode Example for a complete example of a GenerateTree function.

See Also

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-In Functions", Using the GenerateTree Function

GenerateWorklistPortalURL

```
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

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

GenerateWorklistRelativeURL

"Internet Script Classes (iScript)"

PeopleTools PeopleBook: Peoplesoft Workflow: "Introduction to Workflow"

GenerateWorklistRelativeURL

```
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

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

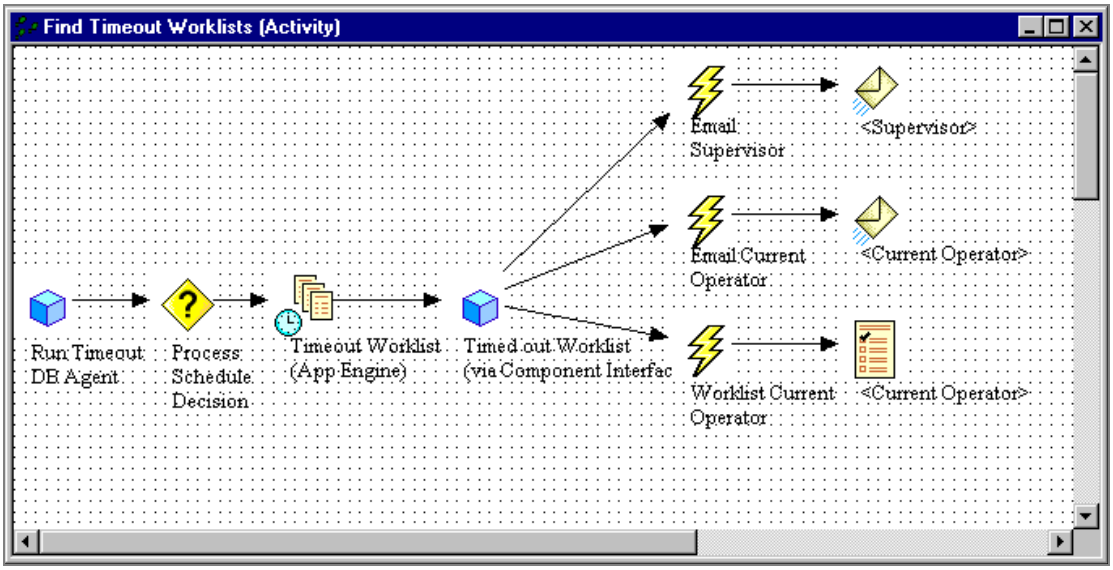
A string of the following format:

```
../../../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

GenerateWorklistPortalURL

"Internet Script Classes (iScript)"

PeopleTools PeopleBook: Peoplesoft Workflow: "Introduction to 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.

See AESection Class for more details.

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.

See How an AESection is Accessed for more details.

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

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.

Example

See AESection Example.

See Also

Open AESection class method

"AESection Class"

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.

Considerations Using GetAttachment, PutAttachment

You cannot execute two attachment calls immediately after each other, that is, running GetAttachment immediately PutAttachment or vice-versa. If you need to do something like this, add the following PeopleCode after the first attachment call, before the second one.

```
&current_time = %Datetime;
&delay_time = AddToDateTime(&current_time, 0, 0, 0, 0, 0, 0);
```

```
While True
   If %Datetime > &delay_time Then
     Break;
   End-If;
End-While;
```

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

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. 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:

Number	Constant	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

AddAttachment, DeleteAttachment, PutAttachment, ViewAttachment

PeopleTools PeopleBook: Peoplesoft 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.

See PeopleSoft Business Interlink Application Developer Guide for more details.

Parameters

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);
```

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

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

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

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

"Chart Class," Creating a Chart

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

&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

"Chart Class"

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

GetEnv, ExpandEnvVar

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.

Parameters

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

GetCwd, ExpandEnvVar

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;
```

Returns

This function returns a field object that references the field from the specified record.

Parameters

recname.fieldname	If you don't want to refer to the field executing the PeopleCode, specify a record name and field name.

Example

```
Local Field &CHARACTER;

&CHARID = GetField(FIELD.CHAR_ID);
```

See Also

"Field Class"

"Record Class," GetField

PeopleTools PeopleBook: Peoplesoft 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

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.

	"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 file class methods to maintain checkpoints of the current read/write position in the file.

Warning! 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_Absoulte

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

FileExists, FindFiles, GetFile

"File Class", Open, GetPosition, SetPosition, IsOpen

"File Class", File Access Interruption Recovery

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 "Grid Classes"

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

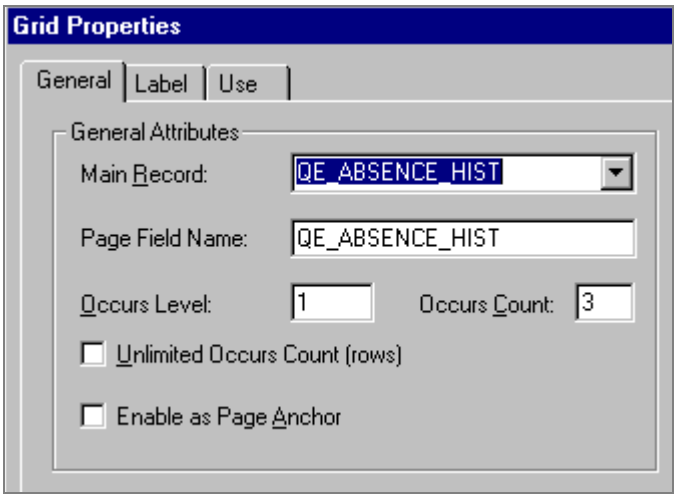
When you place a grid on a page, the gird is automatically named the same as the name of the primary record of the scroll for the grid (in the Page Field Name.)

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.

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.**



Changing a grid name

Note. Every grid on a page must have a unique name.

Parameters

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

"Grid Classes," GetColumn

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 pre-defined 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.

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

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

"Internet Script Classes (iScript)"

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-In Functions," Using HTML Definitions and the GetHTMLText Function

PeopleTools PeopleBook: Application Designer, "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

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

PeopleTools PeopleBook: Peoplesoft Application Designer, "Page Controls," 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.

See Refresh rowset class method.

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 Generating the PeopleCode Template for more details.

Parameters

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

"Business Interlink Class"

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.

See Java Class for more details.

Parameters

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

CreateJavaArray, CreateJavaObject

"Java Class"

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

GetRowset,

"Rowset Class"

PeopleTools PeopleBook: 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

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

"Component Interface Classes"

"Internet Script Classes (iScript)," Web Libraries

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

CreateMessage

"Message Class"

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

PublishXmlDoc, SyncRequestXmlDoc

"XmlDoc Class"

"Message Class"

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.

Parameters

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:

Return	Constant	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

GetNextNumberWithGaps

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
- 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.

Parameters

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:

Return	Constant	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

GetNextNumber

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 False for *Commit*.

Parameters

Commit	Specify whether the current data instance should be committed to the database. This parameter takes a Boolean value: True, commit the data, False, do not commit the data. True 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

ProcessRequest Class"

PeopleTools PeopleBook: Peoplesoft Process Scheduler, "Defining Jobs and JobSets"

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

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

GetMessageXmlDoc

"XmlDoc Class"

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Messages," Understanding Nonrepudiation

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.

See Page Class for more information.

Note

An expression of the form

```
PAGE.name.property
```

is equivalent to **GetPage**(*name*).*property*.

Parameters

PAGE.*page*name	The name of the page for which you want to create an object reference. Must be a page in the current context.

See Understanding Current Context for more information.

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

"Page Class"

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

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

CreateMessage, GetMessage, GetSubContractInstance, GetMessageInstance

"Message Class"

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

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

ReSubmitPubHeaderXmlDoc, GetPubXmlDoc

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

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

ReSubmitPubXmlDoc, ReSubmitPubHeaderXmlDoc

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.

See Understanding Current Context for more information.

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).

Returns

GetRecord returns a record object.

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.

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

CreateRecord

"Record Class"

PeopleTools PeopleBook: Peoplesoft PeopleCode Developer's Guide, "Data Buffer Access"

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. See Data Buffer Access.

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.**

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

PeopleTools PeopleBook: Peoplesoft 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.

See Understanding Current Context for more information.

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 ActiveX controls, 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

GetRowset

"Row Class"

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.

See Understanding Current Context for more information.

Note

An expression of the form

```
RECORD.scrollname.property
```

or

```
RECORD.scrollname.method(…)
```

is converted to an object expression by using GetRowset(**SCROLL.***scrollname*).

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.

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.

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

GetLevel0

"Rowset Class"

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 Appendix B: Viewing Trees From Application Pages.

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.

See Session Class for more information.

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

"Component Interface Classes"

"PortalRegistry Classes"

"Query Classes"

"Search Classes"

"Session Class"

"Tree Classes"

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.

See Control Tables for more information about TableSets.

Returns

GetSetId returns a five-character setID string.

Parameters

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.

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,
"");
```

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 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.

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

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

CreateSQL, DeleteSQL, FetchSQL, SQLExec

"SQL Class," Open

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. To return the format for a field on level zero of the page, pass 1 in *target_row*.

Note. This function remains for backward compatibility only. Use the StoredFormat Field class property instead. See Data Buffer Access.

Returns

Returns a String equal to the name of the stored custom format for the field.

Parameters

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*.

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

SetDisplayFormat

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)

Parameters

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

CreateMessage, GetMessage, GetPubContractInstance, GetMessageInstance

"Message Class"

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

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

ReSubmitSubXmlDoc

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

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

*PeopleTools PeopleBook: 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 Appendix B: Viewing Trees From Application Pages.

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 Appendix B: Viewing Trees From Application Pages.

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 Appendix B: Viewing Trees From Application Pages.

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

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

ViewURL

PeopleTools PeopleBook: Data Management, "PeopleTools Utilities," URL Maintenance

GetUserOption

Syntax

```
GetUserOption(Level, OPTN)
```

Description

Use the **GetUserOption** function to return the default value for the specified option.

Parameters

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

SetUserOption

PeopleTools PeopleBook: Peoplesoft 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

MarkWLItemWorked, TriggerBusinessEvent

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 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.

See Data Types for details about PeopleCode data types.

Note. Because a function can be called from anywhere, you can't declare any variables within a function. You receive a design time error if you try.

Parameters

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

Local, Component

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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 will want to 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:

```
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. See Data Buffer Access.

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.

Event Restrictions

This function shouldn't be used in any event prior to RowInit.

Returns

Optionally returns a Boolean value indicating whether the function succeeded.

Parameters

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*.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

Hide, Ungray, Unhide

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.

PeopleCode Built-in Functions and Language Constructs H-R

Hash

Syntax

Hash(*ClearTextString*)

Description

The Hash function returns a fixed length value, based on the input. The output is always 30 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

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

Decrypt, Encrypt

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

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

CubicSpline, LinearInterp

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

Description

The **Hide** function makes a page field invisible. 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.

Note. This function remains for backward compatibility only. Use the Visible Field class property instead. See Data Buffer Access.

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.

Event Restrictions

This function shouldn't be used in any event prior to RowInit.

Returns

Boolean (optional). **Hide** returns a Boolean value indicating whether it executed successfully.

Parameters

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*.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

Gray, Ungray, Unhide

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

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

DisableMenuItem, EnableMenuItem,

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

Description

HideRow is used to hide a row occurrence programmatically. It hides the specified row and any associated rows at lower scroll levels.

Note. This function remains for backward compatibility only. Use the Visible Row class property instead. See Data Buffer Access.

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.

Returns

Boolean (optional). **HideRow** returns a Boolean value indicating whether the function executed successfully.

Parameters

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

UnhideRow, DeleteRow

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

Description

HideScroll is used to programmatically hide a scroll bar and all data items within the scroll. Typically this function is used in RowInit and FieldChange PeopleCode to modify the page based on user action.

Note. This function remains for backward compatibility only. Use the HideAllRows Rowset class method instead. See Data Buffer Access.

Returns

Boolean. **HideScroll** returns a Boolean value indicating whether the function executed successfully.

Parameters

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

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

UnhideRow, HideRow, UnhideScroll

HistVolatility

Syntax

HistVolatility(*Closing_Prices*, *Trading_Days*)

Description

The **HistVolatility** function computes the historical volatility of a market-traded instrument.

Parameters

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

ConvertRate

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.

Returns

Returns a Number equal to a whole integer value from 0 to 23 representing the hour of the day.

Parameters

time_value	A Datetime or Time value.

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

Minute, Second

Idiv

Syntax

```
Idiv(x, divisor)
```

Description

The **Idiv** function is an explicit integer division operation. It divides one number (*x*) by another (*divisor*).

Parameters

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

Mod, Round, Truncate, Value, Int, Integer

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;
```

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, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.

See Also

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

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:

Number	Constant	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

DeleteImage

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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. See Data Buffer Access.

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.

Parameters

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

DeleteRow, HideRow, UnhideRow

Int

Syntax

```
Int(decimal)
```

Description

Int truncates a decimal number *x* to an integer and returns the result as a Number value.

Returns

Returns a Number equal to *decimal* truncated to a whole integer.

Parameters

decimal	A decimal number to be truncated.

Example

The following example sets &I1 to 1 and &I2 to -4:

```
&I1 = Int(1.975);
&I2 = Int(-4.0001);
```

See Also

Mod, Round, Truncate, Value

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

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

Mod, Round, Truncate, Value, Int

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

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

IsAlphaNumeric, IsDigits, IsDate, IsDateTime, IsNumber, IsTime

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

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

IsNumber, IsAlpha

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

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

IsAlphaNumeric, IsDigits, IsDateTime, IsNumber, IsTime

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

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

IsAlphaNumeric, IsDigits, IsDate, IsNumber, IsTime

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.

See PeopleTools Utilities for information about setting the base time.

Parameters

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

ConvertDatetimeToBase, ConvertTimeToBase, DateTimeToTimeZone, TimeToTimeZone, TimeZoneOffset

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 include sign indicators and comma and period decimal points.

Parameters

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

IsAlpha, IsAlphaNumeric, IsDate, IsDateTime, IsNumber, IsTime

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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. See Data Buffer Access.

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.

Parameters

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

HideRow, UnhideRow

IsMenuItemAuthorized

Syntax

```
IsMenuItemAuthorized(MENUNAME.menuname, BARNAME.barname,
ITEMNAME.menuitem_name, PAGE.pagename [, action])
```

where *action* is one of the following constants:

Constant	Description
%Action_Add	Add
%Action_UpdateDisplay	Update/Display
%Action_UpdateDisplayAll	Update/Display All
%Action_Correction	Correction
%Action_DataEntry	Data Entry
%Action_Prompt	Prompt

If *action* is omitted, the current action is used.

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

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**.

pagename — The name of the page. This parameter must be prefixed with the keyword **PAGE**.

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

Constant	Description
%Action_Prompt	Prompt

Returns

A Boolean value: True if the user is authorized to access the specified page, False otherwise.

See Also

DoModal, DoModalComponent, Transfer, TransferPage

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

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

"Message Class", IsActive

"XmlDoc Class"

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

DoModal, EndModal

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

DoModalComponent

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.

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

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

IsAlpha, IsAlphaNumeric, IsDigits, IsDate, IsDateTime, IsTime

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.

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

SetSearchDialogBehavior

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

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

IsAlpha, IsAlphaNumeric, IsDigits, IsDate, IsDateTime, IsNumber, IsTime

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

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

ExecuteRolePeopleCode, ExecuteRoleQuery, ExecuteRoleWorkflowQuery, IsUserInRole, %PermissionLists, %PrimaryPermissionList, %Roles, %RowSecurityPermissionList

PeopleTools PeopleBook: Peoplesoft Security, "Working With Permission Lists"

IsUserInRole

Syntax

```
IsUserInRole(rolename1 [, rolename2]…)
```

Description

IsInUserClass 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

rolename	An arbitrary-length list of strings, each of which represents a role. PeopleCode Built-in Functions

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 PeopleBook: Peoplesoft 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.

Returns

Returns a String value derived from *source_str*.

Parameters

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.

Example

The following example sets &SHORT_ZIP to "90210":

```
&SHORT_ZIP = Left("90210-4455", 5);
```

See Also

Right

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

Lenb

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

Len

LinearInterp

Syntax

LinearInterp(*DataPoints*)

Description

The **LinearInterp** function computes a set of lines through a sequence of at least two points.

Parameters

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

CubicSpline, HermiteCubic

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

Exp, Log10

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

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

Global, Component

PeopleTools PeopleBook: 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

Exp, Ln

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

Proper, Upper

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

RTrim

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

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

AddEmailAddress, ChangeEmailAddress, DeleteEmailAddress

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

GetWLFieldValue, %WLName

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

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

Min

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.

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 (see 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, ScrollSelectNew, RowScrollSelect, or RowScrollSelectNew function call.

If the *style* parameter specifies a single button (that is, the **OK** button), the function can be called in any PeopleCode event.

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.

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

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.

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.

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.

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.
```

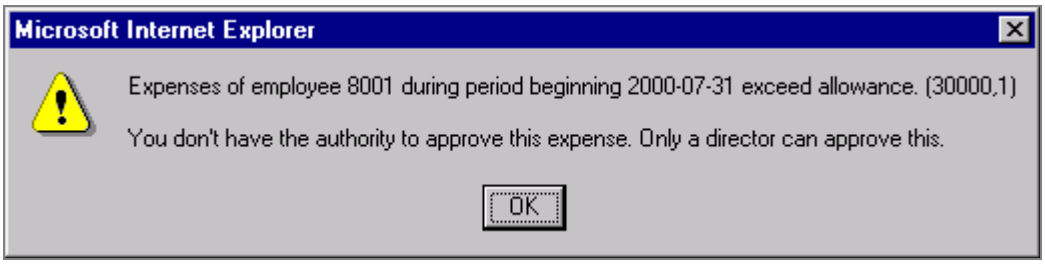
The following is stored in the Explanation:

```
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.
```

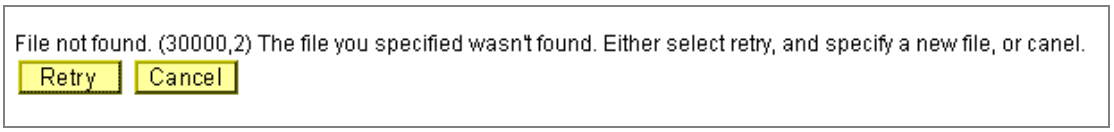
The following is stored in the Explanation:

```
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:

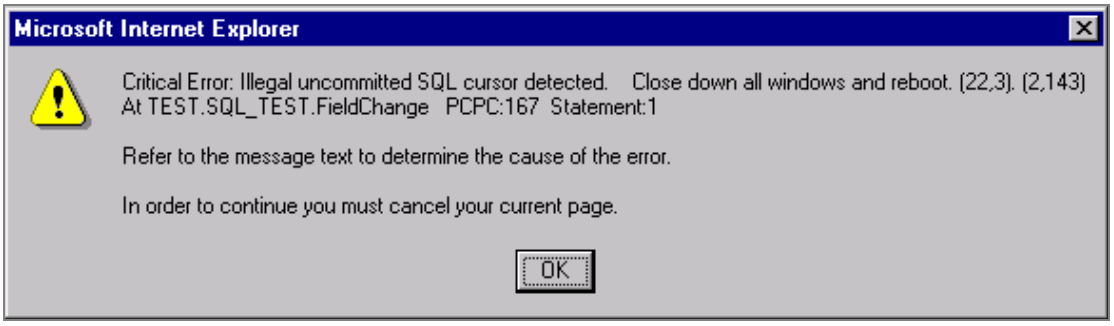
```
MessageBox(%MsgStyle_RetryCancel, "", 30000, 2, "Message not found.");
```

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

MsgGet, MsgGetText, MsgGetExplainText

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

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

Max

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

Hour, Second

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

Int, Round, Truncate

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.

Returns

Returns a Number value from 1 to 12 specifying the month of the year.

Parameters

datevalue	A Date or DateTime value on the basis of which to determine the month.

Example

This example sets &HIRE_MONTH to 3:

```
&HIREDATE = DateTime6(1997, 3, 15, 10, 9, 20);
&HIRE_MONTH = Month(&HIRE_DATE);
```

See Also

Date, Date3, DateValue, Day, Days360, Days365, Weekday, Year

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

MsgGetText, MsgGetExplainText, MessageBox

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

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

MsgGetText, MsgGet, MessageBox

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

MsgGet, MsgGetExplainText, MessageBox

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

NextRelEffDt, PriorRelEffDt, PriorEffDt

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

GetRelField, NextEffDt, PriorRelEffDt, PriorEffDt

NodeDelete

Syntax

NodeDelete(*nodeName*)

Description

Use **NodeDelete** to delete the specified node and all subordinate objects (transactions, node properties, certificates, and so on.)

Note. 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

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

NodeRename, NodeSaveAs

PeopleTools PeopleBook: 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

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

NodeDelete, NodeSaveAs

PeopleTools PeopleBook: 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

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

NodeRename, NodeDelete

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Administering Basic Integrations"

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

All, AllOrNone, OnlyOne, OnlyOneOrNone

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

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 "]".

Flag	Description
a	Bracket negative numbers with "(" and ")".
q	Display zeros as blanks.
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

String, Value

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

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

CreateObject, ObjectGetProperty, ObjectSetProperty, CreateObjectArray, ObjectDoMethodArray

PeopleTools PeopleBook: 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

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

CreateObject, ObjectGetProperty, ObjectSetProperty, CreateObjectArray, ObjectDoMethod

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-In Functions," Using OLE Functions

"Array Class"

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*.

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 retrieve explicitly.

Parameters

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

CreateObject, ObjectDoMethod, ObjectSetProperty, CreateObjectArray, ObjectDoMethodArray

PeopleTools PeopleBook: 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

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

CreateObject, ObjectDoMethod, ObjectGetProperty, CreateObjectArray, ObjectDoMethodArray

PeopleTools PeopleBook: 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

All, AllOrNone, None, OnlyOneOrNone

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

All, AllOrNone, None, OnlyOne

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.

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

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

"Array Class"

"Message Class"

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining 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

NextEffDt, NextRelEffDt, PriorRelEffDt

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

NextEffDt, NextRelEffDt , PriorEffDt

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.

Returns

Returns an Any value equal to the value that was in the current buffer field immediately prior to the last edit.

Parameters

fieldname	The name of the record field. For correct results, this must be the name of the field where the call to **PriorValue** executes.

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

CurrentRowNumber

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

Fact

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

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.

Parameters

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

"Query Classes"

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using PeopleCode Methods and Built-in Functions," Think-Time Functions

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

Lower, Upper

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

&XmlDoc	Specify an already instantiated and populated XmlDoc object that you want to send as an asynchronous message.
Message.*MessageName*	Specify an already existing unstructured message.
Node.*NodeName*	Specify a node. This is for Sender Specified Routing (SSR). 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

GetMessageXmlDoc, SyncRequestXmlDoc

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.

Considerations Using GetAttachment, PutAttachment

You cannot execute two attachment calls immediately after each other, that is, running GetAttachment immediately PutAttachment or vice-versa. If you need to do something like this, add the following PeopleCode after the first attachment call, before the second one.

```
&current_time = %Datetime;
&delay_time = AddToDateTime(&current_time, 0, 0, 0, 0, 0, 0);

While True
   If %Datetime > &delay_time Then
     Break;
   End-If;
End-While;
```

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

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.

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 hard-code path names in the *LocalFile* parameter. This parameter takes a string value.

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:

Number	Constant	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

AddAttachment, DeleteAttachment, GetAttachment, ViewAttachment

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-In Functions," Understanding File Attachment Architecture

Radians

Syntax

```
Radians(angle)
```

Description

Radians converts the given angle from degree measurement to radian measurement.

Parameters

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

Acos, Asin, Atan, Cos, Cot, Degrees, Sin, Tan

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

Int

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)
```

See Specifying Data With References Using Scroll Path Syntax and Dot Notation and Understanding Current Context for more details on scroll path and contextual references.

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. See Data Buffer Access. See also DeleteEnabled Row class property.

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.*

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.

Parameters

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*.**

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

FieldChanged, RecordDeleted, RecordNew

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)
```

See Specifying Data With References Using Scroll Path Syntax and Dot Notation and Understanding Current Context for more details on scroll path and contextual references.

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. **RecordDeleted** is useful during save processing to make processes conditional on whether a row has been deleted.

Note. This function remains for backward compatibility only. Use the IsDeleted Record class property instead. See Data Buffer Access. See also IsDeleted Row property.

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.*

Returns

Returns a Boolean value:

- True if the target row has been deleted.
- False if the target row has not been deleted.

Parameters

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*.**

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

FieldChanged, RecordChanged, RecordNew

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)
```

See Specifying Data With References Using Scroll Path Syntax and Dot Notation and Understanding Current Context for more details on scroll path and contextual references.

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. See Data Buffer Access.

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).

Returns

Returns a Boolean value:

- True if the target row is new.
- False if the target row is not new.

Parameters

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*.**

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

FieldChanged, RecordChanged, RecordDeleted

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 Appendix B: Viewing Trees From Application Pages.

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. GetLevel0().Refresh() refreshes the entire page. If you only want a particular scroll to be redrawn, you can refresh just that part.

See Refresh rowset class method.

Returns

None.

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.
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.

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

Exec, WinExec, and 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

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_Absoulte

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

CreateDirectory, DeleteAttachment, FileExists, FindFiles

"File Class"

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

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:

Constant Value	Description
%MDA_Success	Bulk operation completed successfully.
%MDA_Failure	Bulk operation did not complete successfully.

Constant Value	Description
%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

RenamePage, RenameRecord

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

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:

Constant Value	Description
%MDA_Success	Bulk operation completed successfully.
%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

RenameDBField, RenameRecord

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

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:

Constant 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

RenameDBField, RenamePage

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.

Returns

Returns a String value in which specific characters in *oldtext* are replaced with *newtext*.

Parameters

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.

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

Substitute

Rept

Syntax

```
Rept(str, reps)
```

Description

Rept replicates a text string a specified number of times and combines the result into a single string.

Returns

Returns a String value equal to *str* repeated *reps* times.

Parameters

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.

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

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

ReSubmitPubXmlDoc

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

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

ReSubmitSubXmlDoc, ReSubmitPubHeaderXmlDoc

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

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

ReSubmitPubHeaderXmlDoc, ReSubmitPubXmlDoc

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

Function, Exit

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

TRUE | FALSE | *&NODE_ARRAY*	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

"XmlDoc Class"

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Message Channels"

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

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

GetNRXmlDoc

"XmlDoc Class"

RevalidatePassword

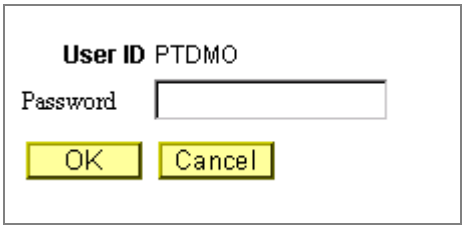
Syntax

```
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, (see Think-Time Functions), 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, ScrollSelectNew, RowScrollSelect or RowScrollSelectNew function call.

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.

Returns

Returns a String value equal to the rightmost *num_chars* character(s) in *str*.

Parameters

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.

Example

If &ZIP is equal to "90210-4455", the following example sets &SUFFIX to "4455":

```
&SUFFIX = Right(&ZIP, 4)
```

See Also

Left

Round

Syntax

```
Round(dec, precision)
```

Description

Use **Round** to round a decimal number to a specified precision.

Returns

Returns a Number value equal to *dec* rounded up to *precision* decimal places.

Parameters

dec	A Number value to be rounded.
precision	A Number value specifying the decimal precision to which to round *dec*.

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

Int, Mod, Truncate

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.

See Controlling Currency Display Format for more details about currencies.

Returns

Returns a Number value equal to *amt* rounded to the currency precision for *currency_cd*.

Parameters

amt	The amount to be rounded.
currency_cd	The currency code.
effdt	The effective date of currency rounding.

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);
```

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

Description

RowFlush is used to remove a specific row from a page scroll and from the component buffer. Rows that are flushed are not deleted from the database.

Note. This function remains for backward compatibility only. Use the FlushRow Rowset method instead. See Data Buffer Access.

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.

Returns

None.

Parameters

scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	The row number of the row to flush.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

Example

The following example flushes a row in a view from the component buffer:

```
RowFlush(RECORD.BNK_RCN_DTL_VW, &ROW1);
```

See Also

ScrollFlush, DeleteRow

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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. See Data Buffer Access.

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 ScrollSelect for more details about 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.

Parameters

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

RowScrollSelectNew, ScrollSelect, ScrollSelectNew, ScrollFlush, SQLExec

Using PeopleCode With PeopleSoft Internet Architecture

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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. See Data Buffer Access.

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.

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.

Parameters

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

RowScrollSelect, ScrollSelect, ScrollSelectNew, ScrollFlush, SQLExec

Using PeopleCode With PeopleSoft Internet Architecture

RTrim

Syntax

```
RTrim(source_str [, trim_str])
```

Description

Use the **RTrim** function to remove characters, usually trailing blanks, from the right of a string.

Returns

Returns a String formed by deleting, from the end of *source_str*, all occurrences of each character specified in *trim_str*.

Parameters

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.

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

LTrim

PeopleCode Built-in Functions and Language Constructs S-Z

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 "ProcessRequest Class".

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. 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.

Note. This function remains for backward compatibility only. Use the Flush Rowset method instead. See *PeopleTools PeopleBook: PeopleCode Developer's Guide,* "Data Buffer Access".

Returns

None.

Parameters

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

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

RowFlush, RowScrollSelect, RowScrollSelectNew, ScrollSelect, ScrollSelectNew

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer", Specifying Data With References Using Scroll Path Syntax and Dot Notation

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 *PeopleTools PeopleBook: PeopleCode Developer's Guide,* "Data Buffer Access".

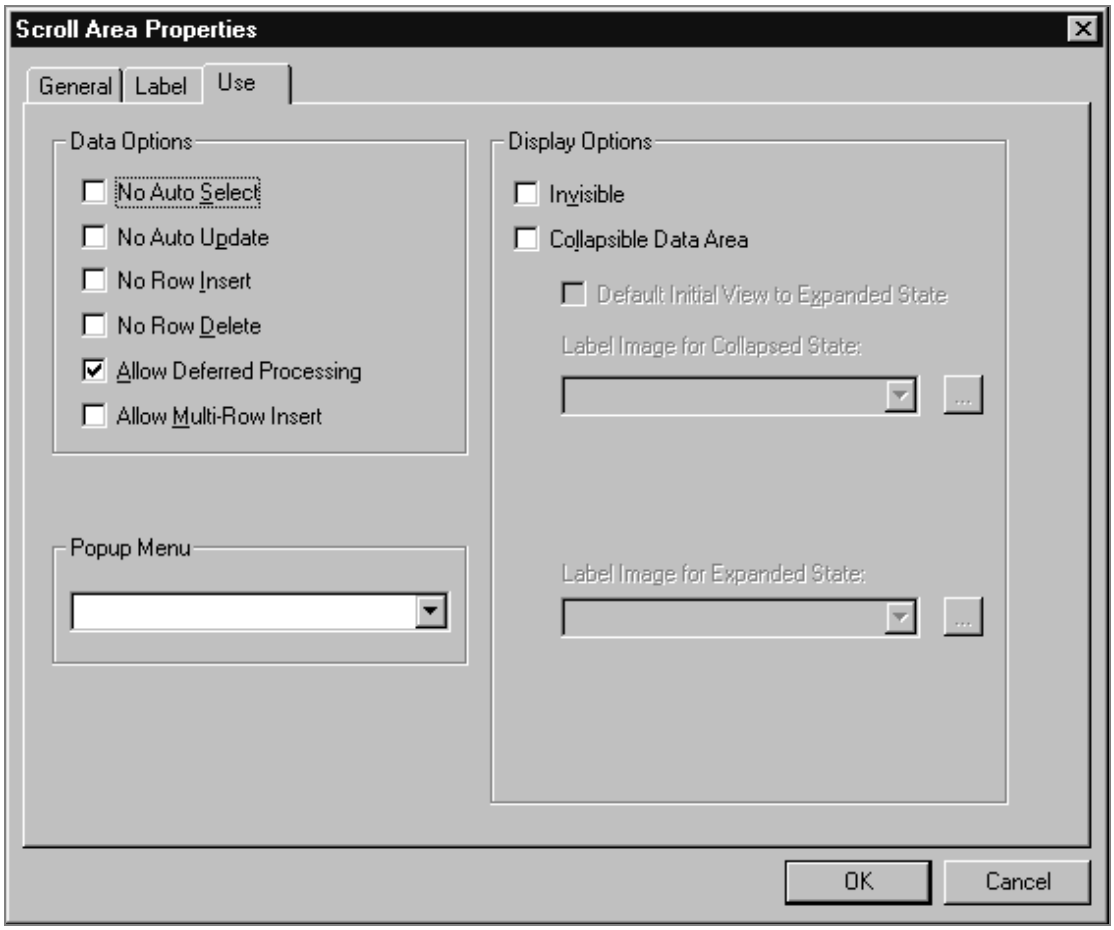
ScrollSelect automatically places child rows in the target scroll area under the correct parent row in the next highest scroll area. If it cannot match a child row to a parent row an error occurs.

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 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 dialog box:



Scroll Area Properties Use tab

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.

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.

Parameters

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.

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

RowScrollSelect, RowScrollSelectNew, ScrollFlush, ScrollSelectNew, SQLExec

Using PeopleCode With PeopleSoft Internet Architecture

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. See Data Buffer Access.

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.

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.

Parameters

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.

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

RowScrollSelect, RowScrollSelectNew, ScrollSelect, ScrollFlush, SQLExec

Using PeopleCode With PeopleSoft Internet Architecture

Second

Syntax

Second(*timevalue*)

Description

Use the **Second** function to extract the seconds component of a Time value.

Returns

Returns a Number equal to the seconds part of *timevalue*.

Parameters

timevalue A Time value from which to extract seconds.

Example

Assume that &TIMEOUT contains Time value of 16:48:21. The following would set &SECS to 21:

```
&SECS = Second(&TIMEOUT);
```

See Also

Hour, Minute

SendMail

Syntax

```
SendMail(flags, recipients, CCs, BCCs, subject, text,
         [, attachment_filenames][, attachment_titles] [, sender])
```

Description

You can use **SendMail** 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.

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.

See Also

TriggerBusinessEvent

PeopleTools PeopleBook: Peoplesoft Workflow, "Defining Event Triggers"

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.

Parameters

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.

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.

sender 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.

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*])

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.

Parameters

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.

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.

ResultDocument — Specify an HTML document explaining the result of the authentication. This document is displayed to the user after either a successful or unsuccessful signon. The default value is null. This parameter takes a string value.

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.

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;
```

See Also

"System Variables," %ResultDocument, %AuthenticationToken

PeopleTools PeopleBook: Peoplesoft Security, "Understanding Peoplesoft Security," PeopleSoft Signon

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

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:

Constant 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

"Field Class", GetAuxFlag

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

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:

Constant 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.

Constant Value	Description
%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

SetDBFieldLabel

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

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:

Constant 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

SetDBFieldFormatLength

"Field Class," StoredFormat, DisplayFormat

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

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:

Constant 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

SetDBFieldFormat

"Field Class," FormatLength

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

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:

Constant Value	Description
%MDA_Success	Function completed successfully.

Constant Value	Description
%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

"Field Class," Label

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

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:

Constant 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

"Field Class," FieldLength

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

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:

Constant 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

PeopleTools PeopleBook: Peoplesoft Application Designer, "Appendix F: Performing Bulk Operations"

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

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

"Message Class"

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

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation.

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.

Returns

None.

Parameters

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.

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

TransferPage

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. See Data Buffer Access.

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.

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.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

[recordname.]fieldname	Specify a field designating the fields that you want to set to its default value.

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

SetDefaultAll, SetDefaultNext, SetDefaultPrior

PeopleTools PeopleBook: PeopleCode Developer's Guide, " PeopleCode and the Component Processor," Default Processing

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 Data Buffer Access and SearchDefault Field method.

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

SetDefault, SetDefaultNext, SetDefaultNextRel, SetDefaultPrior, SetDefaultPriorRel

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

SetDefaultAll, SetDefaultNextRel, SetDefaultPrior, SetDefaultPriorRel

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

SetDefault, SetDefaultAll, SetDefaultPrior, SetDefaultPriorRel

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

SetDefault, SetDefaultAll, SetDefaultNext, SetDefaultNextRel, SetDefaultPriorRel

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

SetDefault, SetDefaultAll, SetDefaultNext, SetDefaultNextRel, SetDefaultPrior

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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 DataAreaCollapsed Field property instead. See Data Buffer Access.

Returns

Returns a Boolean value indicating whether the function executed successfully. The return value is not optional.

Parameters

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

See Also

GetStoredFormat

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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. See Data Buffer Access. See also GetLongLabel, GetShortLabel Field methods.
Note. 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.

Returns

Optionally returns a Boolean value indicating whether the function completed successfully.

Parameters

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.

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 will *not* reflect the value set using SetLanguage until a new component is started.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully. Returns False if an invalid language code is passed.

Parameters

language_code	A valid language code, stored in the Translate table for the LANGUAGE_CD field.

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

"System Variables," %Language

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.

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.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

pagename	A String equal to the name of the page as specified in the page definition.

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( );
```

The following example sets up and transfers the user to page JOB_DATA.

```
If SetNextPage(PAGE.JOB_DATA) Then
     TransferPage( );
End-if;
```

See Also

TransferPage

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

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:

Constant Value	Description
%MDA_Success	Function completed successfully.
%MDA_Failure	Function didn't complete successfully.
%MDA_PageNotFound	The specified page wasn't found.
%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

PeopleTools PeopleBook: 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*.

See Security for more details about password expired status.

Parameters

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

SwitchUser

"System Variables," %PasswordExpired

Security

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

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:

Constant Value	Description
%MDA_Success	Function completed successfully.
%MDA_Failure	Function didn't complete successfully.

Constant Value	Description
%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;
```

See Also

App designer Edit Table info

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

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:

Constant 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

PeopleTools PeopleBook: 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

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. See Data Buffer Access.

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.

Parameters

[*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

ClearSearchDefault, ClearSearchEdit, SetSearchDialogBehavior, SetSearchEdit

PeopleTools PeopleBook: 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.

Returns

None.

Parameters

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**.

Example

The following function call, which must occur in SearchInit PeopleCode, sets the dialog behavior to **skip if possible**.

```
SetSearchDialogBehavior(0);
```

See Also

SetSearchDefault, SetSearchEdit, ClearSearchEdit, ClearSearchDefault, IsUserInRole

PeopleTools PeopleBook: 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 Data Buffer Access.

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.

Returns

Returns a Boolean value indicating whether the function executed successfully.

Parameters

fieldname	The name of the search dialog field on which to enable field edits.

Example

This example turns on edits and system defaults for the SETID field in the search dialog box:

```
SetSearchEdit(SETID);
SetSearchDefault(SETID);
```

See Also

ClearSearchEdit, ClearSearchDefault, SetSearchDefault, SetSearchDialogBehavior

PeopleTools PeopleBook: PeopleCode Developer's Guide, "PeopleCode and the Component Processor," Search Processing in Update Modes

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

instance_number Specify the instance number for the temporary tables.

Returns

Existing (or previous) instance number.

Example

To avoid interferring 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

%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 object properties instead of this function. See Trace Settings.

You can set options prior to starting a PeopleTools session using the Trace tab of PeopleSoft Configuration Manager.

See Understanding 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.

Returns

None.

Parameters

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:

Option	Constants	Description
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.

Option	Constants	Description
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.
2048		Show each statement as it's executed (and don't show statements on branches not taken.)

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

SetTraceSQL

PeopleTools PeopleBook: Data Management, "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 object properties instead of this function. See Trace Settings.

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.

See Understanding 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.

Returns

None.

Parameters

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:

Option	Constant	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

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.

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

SetTracePC

PeopleTools PeopleBook: Data Management, "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

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

"ProcessRequest Class," Schedule

PeopleTools PeopleBook: Process Scheduler, "Understanding PeopleSoft Process Scheduler"

SetUserOption

Syntax

```
SetUserOption(Level, OPTN, Value)
```

Description

Use the **SetUserOption** to set the default value for the specified option.

Parameters

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

GetUserOption

PeopleTools PeopleBook: Peoplesoft Security, "Managing PeopleSoft Personalizations"

Sign

Syntax

```
Sign(n)
```

Description

Sign determines the sign of a number.

Returns

Returns a Number value equal to:

- 1 if *n* is positive
- 0 if *n* is 0
- -1 if *n* is negative

Parameters

n	A Number value of which to determine the sign.

Example

The example sets &NUMSIGN to 1:

```
&NUMSIGN = Sign(25);
```

See Also

Abs

Sin

Syntax

```
Sin(angle)
```

Description

Sin calculates the sine of the given angle (opposite / hypotenuse).

Parameters

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

Acos, Asin, Atan, Cos, Cot, Degrees, Radians, Tan

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.

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.

Parameters

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.

Example

The example calculates &PMT as .857338820301783265:

```
&PMT = SinglePaymentPV(8, 2);
```

See Also

UniformSeriesPV

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. See Data Buffer Access.

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.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

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.

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

HideScroll, RowScrollSelect, RowScrollSelectNew, ScrollSelect, ScrollSelectNew, UnhideScroll

PeopleTools PeopleBook: Data 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

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

CreateArray, CreateArrayRept

"Array Class"

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 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. 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 the example later in this section about how to use meta-SQL. See Meta-SQL for more details about the meta-SQL functions.

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 will 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.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

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.

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

CreateSQL, FetchSQL, GetSQL, StoreSQL

"SQL Class"

Sqrt

Syntax

```
SQRT(n)
```

Description

Sqrt calculates the square root of a number.

Returns

Returns a Number equal to the positive square root of *n*. If *n* is a negative number, **Sqrt** displays an error.

Parameters

n	A Number of which you want to find the square root.

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.

See Also

"Component Interface Classes"

StopFetching

Syntax

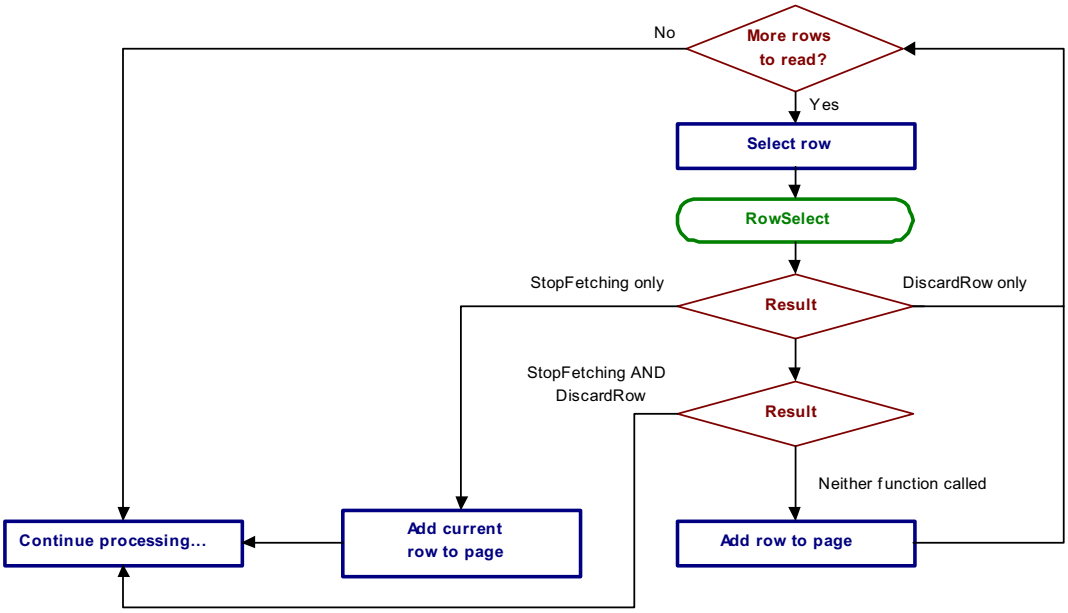
```
StopFetching()
```

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

DiscardRow

PeopleTools PeopleBook: 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

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:

- DEFAULT
- DB2ODBC
- DB2UNIX
- INFORMIX
- MICROSOFT
- ORACLE
- SYBASE

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

CreateSQL, DeleteSQL, FetchSQL, GetSQL, SQLExec, CommitWork

"System Variables," %DbName, %AsOfDate

"SQL Class"

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

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

Char, Exact, Find, Left, Substring, Value, NumberToString

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.

Returns

Returns a String resulting from replacing every occurrence of *old_text* found in *source_text* with *new_text*.

Parameters

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*.

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("abababab", "ab", "cd");
```

See Also

Replace

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.

Returns

Returns a String equal to a substring *length* characters long beginning at character *start* of *source_str*.

Parameters

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.

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

Char, Exact, Find, Left, Right, String, Substringb

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.

Returns

Returns a String value equal to a substring *length* bytes long beginning at byte *start* of *source_str*.

Parameters

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.

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

Char, Exact, Find, Left, Right, String, Substring

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

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.

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	*S*pecify 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

SetPasswordExpired

"System Variables," %UserId, %PasswordExpired

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

&XmlDoc	Specify an already instantiated and populated XmlDoc object that you want to send as a synchronous message.
Message.*MessageName*	Specify an already existing unstructured message.
Node.*NodeName*	Specify a node. This is for Sender Specified Routing (SSR). 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

PublishXmlDoc

"Message Class," SyncRequest

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Messages"

Tan

Syntax

```
Tan(angle)
```

Description

Tan calculates the tangent of the given angle (opposite / adjacent).

Parameters

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

Acos, Asin, Atan, Cos, Cot, Degrees, Radians, Sin

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

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

CreateException, Try

"Exception Class"

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.

Returns

Returns a Time value based on the number *n*.

Parameters

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.

Example

The example sets &START_TIME to 12:34:56.123456:

```
&START_TIME = Time(123456.123456);
```

See Also

Date, DateTimeValue, Time3, TimeValue

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.

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.

Parameters

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.

Example

The example sets &START_TIME to 11.14.09.300000:

```
&START_TIME = Time3(11,14,9.3);
```

See Also

Date3, DateTime6, Time, TimeValue

TimePart

Syntax

```
TimePart(datetime_val)
```

Description

TimePart derives the Time component of a DateTime value.

Returns

Returns a Time value equal to the time portion of *datetime_val*.

Parameters

datetime_val	A DateTime value from which to extract the Time component.

Example

The example set &T to 15.34.35.000000:

```
&DT = DateTimeValue("12/13/93 3:34:35 PM");
&T = TimePart(&DT);
```

See Also

DatePart, Hour, Minute, Second

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

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*.

See PeopleTools Utilities for more details about setting the base time.

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

ConvertDatetimeToBase, ConvertTimeToBase, FormatDateTime, IsDaylightSavings, DateTimeToTimeZone, TimeZoneOffset

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.

Returns

Returns a Time value based on *time_str*.

Parameters

time_str	A String representing a time in the form HH:MM:SS.SSSSSS, based on a 24-hour clock.

Example

The example sets &START_TIME to 12.13.00.000000:

```
&START_TIME = TimeValue("12:13:00 PM");
```

See Also

DateTimeValue, DateValue

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

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

ConvertDatetimeToBase, ConvertTimeToBase, FormatDateTime, IsDaylightSavings, DateTimeToTimeZone, TimeToTimeZone

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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. See Data Buffer Access.

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.

Returns

Returns a Number equal to the total rows (including rows marked as deleted) in the target scroll.

Parameters

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

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

ActiveRowCount, CopyRow, CurrentRowNumber, FetchValue, For

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.

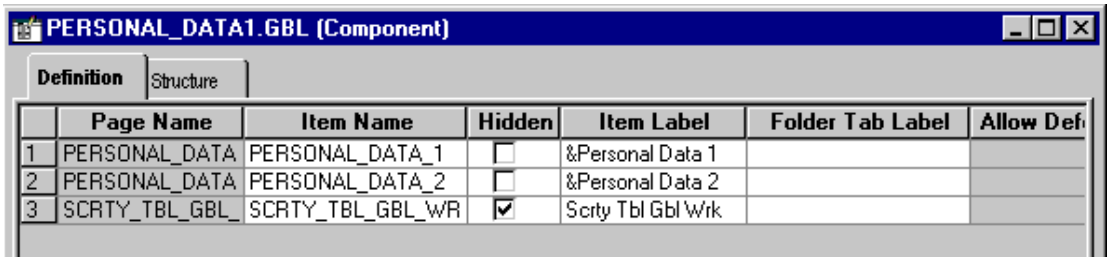
You can use **Transfer** from a secondary page (either with or without using a pop-up menu) *only* if you're transferring to a separate instance of a component. You *cannot* use **Transfer** from a secondary page if you're not transferring to a separate instance of a component.

If you provide a valid search key for the new page in the optional *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 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. See Creating Component Definitions.

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

See Also

TransferPage

PeopleTools PeopleBook: Application Designer, "Creating Menu Definitions"

PeopleTools PeopleBook: 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.

Parameters

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

TransferPage, DoModalComponent

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. See Creating Component Definitions.

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.

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

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

TransferPortal, TransferPage, Transfer

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. See Creating Component Definitions.

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.

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

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

TransferNode, TransferPage, Transfer

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.

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.

Note. You can't use **TransferPage** from a secondary page.

TransferPage stops all further processing (including the program containing this function), unless you are in the middle of save processing. If **TransferPage** is called in SavePreChange or SavePostChange PeopleCode, the processing completes before the transfer is processed.

If **TransferPage** 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.**TransferPage** can be used without a *page_name* parameter, but the page must have been previously set with SetNextPage.

Any variable declared as a Component variable will still be defined after using a **TransferPage** function.

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

Page.*page_name*	A String equal to the name of the page you are transferring to, as set in the page definition. 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

DoModalComponent, SetNextPage, Transfer

Transform

Syntax

```
Transform(Input, 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.

Parameters

Input	Specify 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 if successful, NULL otherwise.

See Also

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Applying Transformation, Translation and Filtering"

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

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

"Tree Classes"

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.

See Adding Events for more details about defining business events.

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.

Parameters

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.

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

GetTreeNodeValue, MarkWLItemWorked

PeopleTools PeopleBook: Peoplesoft Workflow, "Defining Event Triggers"

Truncate

Syntax

```
Truncate(dec, digits)
```

Description

Truncate truncates a decimal number *dec* to a specified precision.

Returns

Returns a Number value equal to *dec* truncated to a *digits* precision.

Parameters

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).

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

Int, Mod, Round

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

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

Throw, CreateException

"Exception Class"

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

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

EncodeURL, EncodeURLForQueryString

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

Generally, you want to put the 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.

Description

Ungray makes a gray (non-editable) page field editable, if the field was grayed with a call to the **Gray** function. If the page field is made display-only in the Page Field Properties dialog, then **Ungray** has no effect.

Note. This function remains for backward compatibility only. Use the Enabled Field property instead. See Data Buffer Access.

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.

Event Restrictions

This function shouldn't be used in any event prior to RowInit.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

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*.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

Gray, Hide, Unhide

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

Generally, you want to put the 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.

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. See Data Buffer Access.

Event Restrictions

This function shouldn't be used in any event prior to RowInit.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

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*.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

Gray, Hide, Ungray

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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. See Data Buffer Access.

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.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

scrollpath	A construction that specifies a scroll level in the component buffer.
target_row	An integer specifying which row in the scroll to unhide.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

Example

This example uses a For loop to unhide rows previously hidden with the Hide function. Note that the loop counts backward to assure that renumbering of rows does not affect the loop:

```
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

HideRow

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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 Data Buffer Access.

Returns

Optionally returns a Boolean value indicating whether the function executed successfully.

Parameters

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

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

HideScroll, RowScrollSelect, RowScrollSelectNew, ScrollSelect, ScrollSelectNew, SortScroll

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.

Returns

Returns a Number equal to the value of a single unit after *n_per* payments at an interest rate of *int_rate*.

Parameters

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.

Example

The example sets &NUM to 3.790786769408448256:

```
&NUM = UniformSeriesPV(10,5);
```

See Also

SinglePaymentPV

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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 Data Buffer Access.

Returns

None.

Parameters

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.

See Specifying Data With References Using Scroll Path Syntax and Dot Notation for details about scroll path syntax.

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

FetchValue, PriorValue

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

&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

"XmlDoc Class"

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.

Returns

Returns a String value equal to *str* converted to all uppercase.

Parameters

str	A String to convert to 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

Lower, Proper

Value

Syntax

Value(*str*)

Description

Value converts a string representing a number to the number.

Returns

Returns the Number value represented by *str*.

Parameters

str	A String value representing a number.

Example

The example sets &VAL1 to 5.25 and &VAL2 to 12500:

```
&VAL1 = Value("5.25");
&VAL2 = Value("12,500");
```

See Also

String

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.

Note. .exe and .bat files can be launched instead of 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 Select or SelectNew method, or any of the ScrollSelect functions.

See 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

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.

UserFile	The name of the file on the source system.

Returns

An integer value. You can check for either the integer or the constant:

Number	Constant	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.

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

AddAttachment, DeleteAttachment, GetAttachment, PutAttachment

PeopleTools PeopleBook: 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

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

GetURL, ViewURL

PeopleTools PeopleBook: Data 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. **Note.** Portal applications should use the RedirectURL Response class method instead of ViewURL.

Restrictions on Use With Windows Client

The *ReplaceCurrentPage* parameter is not available for Windows Client.

Parameters

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

GetURL, ViewContentURL

PeopleTools PeopleBook: Data 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.

Returns

None.

Parameters

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.

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.

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.

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.

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

Error, MsgGet, MsgGetText, WinMessage

Weekday

Syntax

```
Weekday(dt)
```

Description

Weekday calculates the day of the week based on a date value.

Returns

Returns a Number value representing the day of the week. 1 is Sunday, 7 is Saturday.

Parameters

dt	A Date value. **Weekday** determines the day of the week that *dt* falls on.

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

Date, Date3, DateValue, Day, Days360, Days365, Month, Year

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

Repeat

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.

See PeopleCode and PeopleSoft Internet Architecture.

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

EndModal, Error, Warning, DoCancel

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.

Returns

WinExec returns a Number indicating whether the function completed successfully:

Error	Description
0	Completed successfully
non-zero	Error

Parameters

window_option Specifies the style of the newly created window. You can specify either a numeric value or a constant:

Option	Constant	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

Option	Constant	Description
7	%WinExec_MaximizedNoFocus	Maximized window without focus

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.

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, ScrollSelectNew, RowScrollSelect, or RowScrollSelectNew function call.

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 PeopleCode and PeopleSoft Internet Architecture.

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

Declare Function , Exec, RemoteCall

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.

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

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 (see 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, ScrollSelectNew, RowScrollSelect, or RowScrollSelectNew function call.

If the *style* parameter specifies a single button (that is, the OK button), the function can be called in any PeopleCode event.

Note. In Windows Client, MessageBox dialogs include an Explain button to display more detailed information stored in the Message Catalog. The presence of the Explain button has no bearing on whether a message box behaves as a think-time function.

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.

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

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. **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.

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.

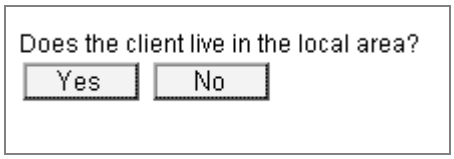
If the *style* parameter is provided, **WinMessage** returns one of the following Number values:

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.
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 WinMessage dialog 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 WinMessage dialog box has no Cancel button, pressing ESC has no effect.

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

Encrypt, MessageBox, MsgGet, MsgGetText, MsgGetExplainText

Year

Syntax

```
Year(dt)
```

Description

Year derives the year component of a Date value.

Returns

Returns a Number value between 1900 and 2078 equal to the year component in *dt*.

Parameters

dt	A Date value from which to derive the year component.

Example

The example sets &GRAD_YEAR to 1976:

```
&GRAD_DATE = DateValue("10/04/76");
&GRAD_YEAR = Year(&GRAD_DATE);
```

See Also

Date, Date3, DateValue, Day, Days360, Days365, Month, Weekday

CHAPTER 2

Meta-SQL

Meta-SQL functions expand to platform-specific SQL substrings. They are used in functions that pass SQL strings, that is:

- 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

This chapter includes the following topics:

- Date Considerations
- Meta-SQL Placement Considerations
- Meta-SQL Reference

See Also

"PeopleCode Built-in Functions", SQLExec, ScrollSelect

"Rowset Class"

"SQL Class"

"Record Class"

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 constructs can be used by all programs. Some meta-SQL constructs can be used in Application Engine. Other constructs 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 constructs. An X in the appropriate column means that you can use that construct in that type of program.

If a meta-SQL construct 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 a meta-SQL statement like a function. It means that you can use the meta-SQL statement 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.

The meta-SQL constructs that are only available for Application Engine are fully described in Application Engine Meta-SQL.

See Also

PeopleTools PeopleBook: Peoplesoft Application Engine, "Using Meta-SQL and PeopleCode"

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**		
%ClearCursor		**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**

Name	*Used in PeopleCode*	*Used in Application Engine*	*Used in COBOL*	*Used in Dynamic View*
%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
%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		
%Next and %Previous		X		
%NoUppercase	X	X		X

Name	*Used in PeopleCode*	*Used in Application Engine*	*Used in COBOL*	*Used in Dynamic View*
%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
%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

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

"PeopleCode Built-in Functions," PeopleCode Typographical Conventions

%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

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

"PeopleCode Built-in Functions," RemoteCall

PeopleTools PeopleBook: Peoplesoft Global Technology, "COBOL Globalization"

%Concat

Syntax

```
string1 %Concat string2
```

Description

At runtime, the **%Concat** function is replaced by the string concatenation operator appropriate for the RDBMS being executed on. For example, on DB2, the %Concat function is replaced with CONCAT, while on Sybase it's replaced with a "+".

This function is supported with the same limitations as the native concatenation operator on the RDBMS the function is being executed on. For example, some platforms will allow you to concatenate a string with a numeric value; others will 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.

Examples

```
SELECT 'A' %Concat 'B' FROM PS_INSTALLATION. . .

SELECT LAST_NAME %Concat ',' %Concat FIRST_NAME FROM PS_EMPLOYEE
```

%CurrentDateIn

%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

%CurrentDateOut expands to platform-specific SQL for the current date in the SELECT clause of an SQL query.

%CurrentDateTimeIn

%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

%CurrentDateTimeOut expands to platform-specific SQL for the current datetime in the SELECT clause of an SQL query.

%CurrentTimeIn

%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

%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.

Examples

```
&ANSWER = %DateAdd(%DateIn('1997-02-10'), 5);

(&ANSWER = 1997-02-15)

&ANSWER = %DateAdd(%DateIn('1997-02-10'), -5);

(&ANSWER = 1997-02-05)

%DateAdd(:1, :2) (:1 and :2 are bind variables in SQLExec PeopleCode)
```

%DateDiff

Syntax

%DateDiff(*date_from*, *date_to*)

Description

%DateDiff returns an integer representing the difference between two dates in number of days.

Examples

```
%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

%DateIn(*dt*), where ***dt*** is either a Date value or a date literal in YYYY-MM-DD format, 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 function 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 LASTUPDT = %DATEIN('2002-12-11')
```

Whereas the following SQL will fail:

```
UPDATE PS_PERSONAL_DATA SET LASTUPDT = %DATEIN(:1)
```

%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

%DateOut(*dt*), where ***dt*** is a date column, expands to a platform-specific SQL substring representing ***dt*** in the SELECT clause of an SQL query.

%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

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

%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. The parameter ***dtt*** is either a Datetime bind variable or a String literal in the form:

```
YYYY-MM-DD-hh.mm.ss.ssssss
```

Restrictions using COBOL

You can only use string literals when using this function 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 LASTUPDTTM = %DATETIMEIN('2002-12-11-
11.59.00.000000')
```

Whereas the following SQL will fail:

```
UPDATE PS_PERSONAL_DATA SET LASTUPDTTM = %DATETIMEIN(:1)
```

%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

%DateTimeOut(*datetime_col*), where ***datetime_col*** is a Datetime column, expands to a platform-specific SQL substring representing ***datetime_col*** in the SELECT clause of an SQL query.

%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

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

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** function 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 effective status (EFF_STATUS) checking.

Parameters

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 *can't* 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

- This function is *not* implemented for COBOL or Dynamic View SQL.
- This function is *not* implemented for Sybase. It will do nothing on this platform.
- This function *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 function 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 function must not be used with DISTINCT, because SELECT TOP 1 DISTINCT will fail on SqlServer.
- This function is implicitly embedded in all SELECT SqlExecs for all platforms except ORACLE.

Parameters

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** function 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

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** function 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

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** function to dynamically build a WHERE clause joining one table to another. At runtime, the entire function will be replaced with a character string.

Note. This meta-SQL is not implemented for COBOL.

Parameters

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*.

Examples

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** function 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

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** function 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

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** function 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. See Considerations using Bind Markers.

%Like generates the following:

```
like 'literal%'
```

If the literal value contains '_' or '\%', %Like generates the following:

```
like 'literal%' escape '\'
```

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	_	%
SQLBase	_	%

DBMS	Single Character	Multiple Characters
Sybase	_	%

Parameters

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

ExpandSqlBinds

Parameters

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** function 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 function.

Note. This meta-SQL is not implemented for COBOL, Dynamic View SQL or PeopleCode.

Parameters

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 *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 *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

Syntaxc

%ListBind({FIELD_LIST | FIELD_LIST_NOLONGS | KEY_FIELDS}, *recordname* [*State_record_alias*])

Description

The **%ListBind** function 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.

Parameters

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 *can't* 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.

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.

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** function 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 function 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

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 *can't* 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)
```

%OldKeyEqual

Syntax

%OldKeyEqual(*recname [correlation_id]*)

Description

The **%OldKeyEqual** function is similar to the **%KeyEqual** function, 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 function 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

%KeyEqual

%OPRCLAUSE

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

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.

Examples

```
%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** function 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. **Note.** 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 will occur.

See Also

"SQL Class"

PeopleTools PeopleBook, 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 constructs can be used as part of the %SQL function 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

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

"SQL Class"

%Substring

%Substring (*source_str*, *start*, *length*) expands to a substring of *source_str*, where *start* specifies the substring's beginning position (the first character of *source_str* is position 1), and *length* is the length of the substring.

%SUBREC

%SUBREC(*subrec_name*, *corel_name*) 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. *subrec_name* is the name of the subrecord; *corel_name* is the correlation name.

%SUBREC must be either all upper case or all lowercase. Mixed case isn't allowed.

For 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** function 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 function can be used to specify temporary tables for running parallel Application Engine processes.

See Also

PeopleTools PeopleBook, Peoplesoft Application Engine, "Using Meta-SQL and PeopleCode," %Table

Parameters

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 *can't* 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

"PeopleCode Built-in Functions," SetTempTableInstance

%Test

Syntax

%Test(*Prefix*, *Test*, *Suffix*)

Description

The **%Test** function can be used with records that have no key values.

Parameters

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, then the entire %Test meta-SQL function is replaced with nothing. Otherwise the %Test meta-SQL function 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 function, 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. **Note.** 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

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 function 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). **Note.** This meta-SQL is not implemented for COBOL.

Parameters

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

%TimeIn(*tm*) 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. The parameter ***tm*** is either a Time bind variable or a String literal in the form:

```
hh.mm.ss.ssssss
```

Restrictions using COBOL

You can only use string literals when using this function 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)
```

%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

%TimeOut(*time_col*), where ***time_col*** is a time column, expands to a platform-specific SQL substring representing ***time_col*** in the SELECT clause of an SQL query.

%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

DTTM_Column	Specify the datetime column you want just the time part returned.

%TrimSubstr

%TrimSubstr(*source_str*, *start*, *length*) is similar to %Substring, except that trailing blanks are removed from the substring.

See Also

%Substring

%Truncate

Syntax

```
%Truncate(expression, factor)
```

Description

%Truncate truncates an expression to a specified scale before or after the decimal point.

Parameters

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.

Examples

```
%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

```
%Truncate(10.337, 2 + 1)  (factor can not be an expression)
```

You may get wrong results or runtime SQL errors when using the illegal case.

%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**.

Example

```
%TruncateTable(PS_TEMP_TABLE)
```

%UpdatePairs

Syntax

```
%UpdatePairs(recname [correlation_id])
```

Description

The **%UpdatePairs** function 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

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** function 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-UPDESCR      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

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

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

Security

%AllowRecipientLookup

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 PeopleBook: Security, "Chapter Name"

%AsOfDate

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

This system variable returns a single signon authentication token for the user after SwitchUser is executed. For example, you can use this system variable to write a single signon cookie to the http response after a new user is authenticated.

Note. This system variable returns a valid value only after SwitchUser executes successfully.
Note. 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

"PeopleCode Built-in Functions," SwitchUser

"Internet Script (iScript) Classes," AuthTokenDomain

%BPName

%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

%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 PeopleBook: Security, "Managing PeopleSoft Personalizations"

%ClientTimeZone

%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 PeopleBook: Security, "Managing PeopleSoft Personalizations"

%Component

%Component returns an uppercase character string containing the name of the current component, as set in the component definition.

%CompIntfcName

%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.)

%ContentID

%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

%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_SE
F.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

This system variable returns a string suitable for use as a standard PeopleSoft copyright notice.

%Currency

This system variable returns the preferred currency for the current user.

%Date

%Date returns a Date value equal to the current server date.

%DateTime

%DateTime returns the current server date and time as a Datetime value.

%DbName

%DbName returns the name of the current database as a String value.

%DbServerName

%DbServerName returns the name of the current Sybase or Informix database server as a string. This is not valid for other database types.

%DbType

%DbType returns a string representing the type of the current database; for example, "MICROSFT" or "SYBASE".

%EmailAddress

This system variable returns the email address of the current user.

%EmployeeId

%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

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 signon PeopleCode as a string using this system variable.

Note. This system variable can be used only in Signon PeopleCode. **Note.** This system variable isn't applicable with the PeopleSoft Internet Architecture.

%FilePath

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

PeopleTools PeopleBook: Peoplesoft Application Engine, "Managing Application Engine Programs," Using the Command Line

%HPTabName

This system variable returns the name of the last homepage tab visited by the user as a string.

%Import

%Import returns True if an import is being performed by PeopleSoft Import Manager and False if not.

%IsMultiLanguageEnabled

This system variable returns True if the current user is multi-language enabled.

See Also

PeopleTools PeopleBook: Peoplesoft Global Technologies, "Working With Language-Sensitive Application Data"

%Language

%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

SetLanguage

%Language_Base

%Language_Base returns the base language for the current database, as set with the PeopleTools Options page.

See Also

PeopleTools PeopleBook: Data Management, "PeopleTools Utilities," PeopleTools Options

%LocalNode

%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_SE
F.GBL
```

This system variable returns the following:

```
hrms
```

See Also

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Administering Basic Integrations"

%Market

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

PeopleTools PeopleBook: Peoplesoft Application Designer, "Creating Component Definitions"

%MaxInterlinkSize

%MaxInterlinkSize returns the current size limit of Interlink objects, as set on the PeopleTools Options page.

See Also

PeopleTools PeopleBook: Data Management, "PeopleTools Utilities," PeopleTools Options

%MaxMessageSize

%MaxMessageSize returns the current size limit of messages, as set on the PeopleTools Options page.

See Also

PeopleTools PeopleBook: Data Management, "PeopleTools Utilities," PeopleTools Options

%Menu

%Menu returns an uppercase string containing the current menu name. It can be used to restrict edits or processing to a specific menu.

%Mode

%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:

Value	*Constant*	*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

%NavigatorHomePermissionList

This system variable returns the navigator homepage permission list for the current user.

%Node

%Node returns the name of the node from the current request object.

See Also

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Administering Basic Integrations"

%OperatorClass

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.

%OperatorId

%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.

%OperatorRowLevelSecurityClass

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.

%OutDestFormat

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

PeopleTools PeopleBook: Peoplesoft Application Engine, "Managing Application Engine Programs," Using the Command Line

%OutDestType

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

PeopleTools PeopleBook: Peoplesoft Application Engine, "Managing Application Engine Programs," Using the Command Line

%Page

%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

%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.

%PanelGroup

%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.

%PasswordExpired

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

"PeopleCode Built-in Functions," SwitchUser

%PermissionLists

This system variable returns an array object containing entries for all the permission lists to which the current user belongs.

%Portal

%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_SE
F.GBL
```

This system variable returns the following:

```
eprocurement
```

See Also

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Administering Basic Integrations"

%PrimaryPermissionList

This system variable returns a string representing the primary permission list of the current user.

%ProcessProfilePermissionList

This system variable returns the process profile Permission List for the current user.

%PSAuthResult

This system variable returns the result (True of False) of PeopleSoft ID and password authentication for the user signing on.

%Request

%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

"Internet Script Classes (iScript)"

%Response

%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

"Internet Script Classes (iScript)"

%ResultDocument

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 signon process (or Signon PeopleCode) to the user.

Note. This system variable can be used only in Signon PeopleCode.

See Also

"PeopleCode Built-in Functions," SwitchUser

%Roles

This system variable returns an array object containing entries for all the roles to which the current user belongs.

%RowSecurityPermissionList

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

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

%ServerTimeZone returns the current time zone on the server as a three-character string.

%Session

%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.

```
Local ApiObject &MySession

&MySession = %Session;
If Not (&MySession) Then
     /* Application level error handling */
End-If;
```

%SignonUserId

%SignonUserId returns the value the user typed in at the signon page.

Note. This system variable can be used only in Signon PeopleCode.

%SignOnUserPswd

%SignOnUserPswd returns the value the user typed in at the signon 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

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

PeopleTools PeopleBook: Peoplesoft Workflow, "Designing Blackberry Email Responses"

%SMTPGuaranteed

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

PeopleTools PeopleBook: Peoplesoft Workflow, "Designing Blackberry Email Responses"

"PeopleCode Built-in Functions", SendMail

"Notification Classes," Send

%SMTPSender

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

PeopleTools PeopleBook: Peoplesoft Workflow, "Designing Blackberry Email Responses"

PeopleTools PeopleBook: Workflow, "Notification Templates"

%SQLRows

%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

This system variable returns a reference to the SyncServer object. If you don't have a current SyncServer object, %SyncServer returns NULL.

See Also

PeopleTools PeopleBook: Peoplesoft Mobile Agent, "Synchronizable Component Interfaces"

%Time

%Time retrieves the current server time.

TransformData

This system variable returns a reference to the TransformData object. If you don't have a current TransformData object, %TransformData returns Null.

See Also

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Applying Transformation, Translation and Filtering"

%UserDescription

This system variable returns the description (if any) listed for the current user.

%UserId

%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

%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

%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

%JavaScript

"Internet Script (iScript) Classes," GetJavaScriptURL

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

"PeopleCode Built-in Functions," PeopleCode Typographical Conventions

%Appserver

At runtime %Appserver is replaced with the name of the application server.

This metavariable is valid for use in attached auxiliary files.

%AppsRel

At runtime %AppsRel is replaced with the application release string.

This metavariable is valid for use in attached auxiliary files.

%Browser

At runtime %Browser is replaced with the browser name as specified by the browser loading the current page.

%BrowserPlatform

At runtime %BrowserPlatform is replaced with the operating system name as specified by the browser loading the current page.

%BrowserVersion

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

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

At runtime %Component is replaced with the component name of the current component.

%Copyright

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

At runtime %DBName is replaced with the name of the application database.

This metavariable is valid for use in attached auxiliary files.

%DBType

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

anything	Specify the string to be encoded.

This metafunction is valid for use in attached auxiliary files.

%Formname

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

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

HTMLDefinition	Specify the name of an HTML definition that contains a JavaScript program.

Example

```
<script src='%JavaScript(PT_EDITSCRIPTS)'></script>
```

%LabelTag

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

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

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

At runtime %Page is replaced by the name of the current page.

%ServicePack

At runtime %ServicePack is replaced with the application service pack string.

This metavariable is valid for use in attached auxiliary files.

%SubmitScriptName

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

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.

See Also

PeopleTools PeopleBook: Data Management, "PeopleTools Utilities," URL Maintenance

Parameters

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>
```

%UserId

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.
- Declaring 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

"SQL Class"

"Meta-SQL", %SQL

PeopleTools PeopleBook: 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.

Declaring 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

PeopleTools PeopleBook: Peoplesoft Application Engine, "Creating Application Engine Programs," Call Section Actions

AESection Class Built-in Function

GetAESection

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

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 AESection Example.

See Also

SetSQL

Close

Syntax

```
Close()
```

Description

Close closes the AESection object. Any unsaved changes to the section are discarded.

Parameters

None.

Returns

None.

Example

See AESection Example.

See Also

Save, Open

"PeopleCode Built-in Functions," GetAESection

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

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

IsOpen

How an AESection is Accessed

"PeopleCode Built-in Functions," GetAESection

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 AESection Example.

See Also

Close, Open

"PeopleCode Built-in Functions," CommitWork

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

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 AESection Example.

See Also

AddStep

"SQL Class"

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

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 AESection Example.

See Also

Open

How an AESection is Accessed

"PeopleCode Built-in Functions," GetAESection

AESection Class Property

In this section, we discuss the IsOpen AESection property.

IsOpen

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.
- 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

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

PeopleTools PeopleBook: 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

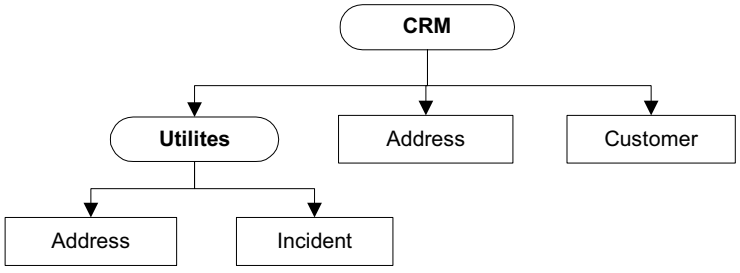
PeopleTools PeopleBook: PeopleCode Developer's Guide, "PeopleCode Language," Data Types

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

Import Declarations

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, property, or instance variable that is declared before the keyword **Private** in the initial class declaration is public. Any method, property or instance variable declared after the keyword **Private** is 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.

The references to the current context and buffers are processed the same as with the current PeopleCode functions and programs. However, there are some differences in how the application class programs are processed by the system:

- The system *never* skips to the next top-level statement.
- The parameter passing conventions are not the same as for existing functions.

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.

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*.

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.

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

PeopleTools PeopleBook: 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 unambiguous, 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");
```

Each subclass is responsible for initializing its superobject. 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.

A class that does not extend some other class does not need to have any constructor.

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

Superclass Reference

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;
```

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;
```

Property Overrides

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 don't 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" */
```

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.

See Also

"Exception Class"

"PeopleCode Built-in Functions," CreateException, Throw, Try

Declaration of 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.

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

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

Method, Get, Set

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

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

Method, Set

"PeopleCode Built-in Functions," Return

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

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

Class

Method

Syntax

```
Method MethodName
     StatementList
End-Method
```

Description

Use the **Method** statement to define the methods of your application class.

Parameters

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

Get, Set

"PeopleCode Built-in Functions," Return

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

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

Get, Method

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

"PeopleCode Built-in Functions," ReturnToServer

PeopleTools PeopleBook: 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

CreateArray

CreateArrayAny

CreateArrayRept

Split

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

Subarray

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

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

Replace, Substitute

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

index	The array element to be accessed.

Returns

An element in an array.

See Also

Set

"Java Class"

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

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

Split

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

&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

Len

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

Push, Replace, Shift, Unshift

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

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

Pop, Replace, Shift, Unshift

Using Flattening and Promotion

PeopleTools PeopleBook: 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 Also

Using Flattening and Promotion

Parameters

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.

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

Substitute, Find

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

Sort

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

index	The array element to be accessed.

Returns

None.

See Also

Get

"Java Class"

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

Pop, Push, Replace, Unshift

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 omitted or "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

order — Specifies whether the array should be sorted in ascending or descending order. If no value is specified for *order* the order is ascending. 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

Array &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

Array &A expanded in PeopleCode debugger, showing code results

See Also

Reverse

PeopleTools PeopleBook: Data Management, "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

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

Clone

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

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

Find, Replace

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

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

Pop, Push, Replace, Shift

Using Flattening and Promotion

Array Class Properties

In this section, we discuss the array class properties.

Dimension

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

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

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

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 Also

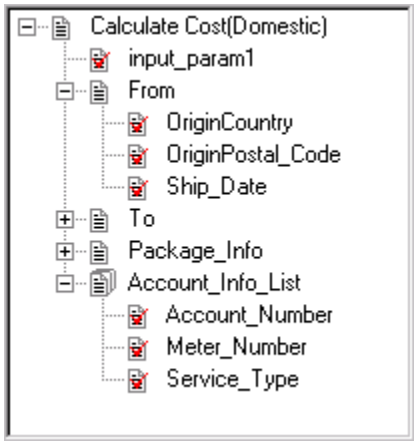
GetFieldCount, GetFieldType, GetFieldValue, MoveFirst, MoveNext

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");
```

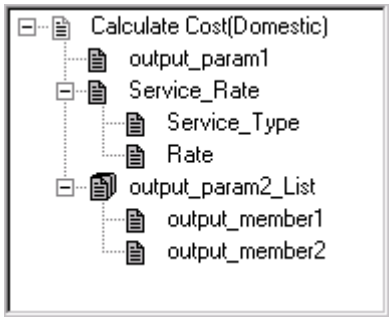
```
&ret = &Account_Doc.AddValue("Account_Number", "CT-8001");
&ret = &Account_Doc.AddValue("Meter_Number", &METER);
&ret = & Account_Doc.AddValue("Service_Type", &MODE);

/* 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");
```

See Also

GetInputDocs, AddDoc, AddValue, AddNextDoc**,** GetOutputDocs, GetDoc, GetNextDoc, GetPreviousDoc, GetStatus, GetValue, GetCount, MoveToDoc

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

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 Function

See Also

"PeopleCode Built-In Functions" GetBiDoc

"PeopleCode Built-In Functions" GetInterlink

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

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;

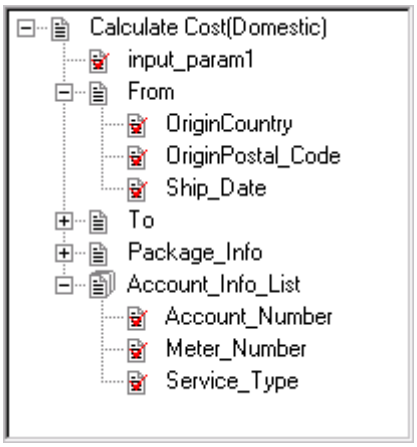

&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);

&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

GetInputDocs, AddValue, AddNextDoc

AddInputRow

Syntax

```
AddInputRow(inputname, value)
```

Description

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 *to* the specified input names (*inputname)* 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 *inputname* 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

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

Execute

"PeopleTools Built-In Functions," GetInterlink

PeopleTools Business Interlink Application Developer Guide PeopleBook, "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

Number	*Enum value and Meaning*
0	eNoError. The method succeeded.
1	eNO_DOCUMENT. The document referenced by this method does not exist.

Number	*Enum value and Meaning*
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;

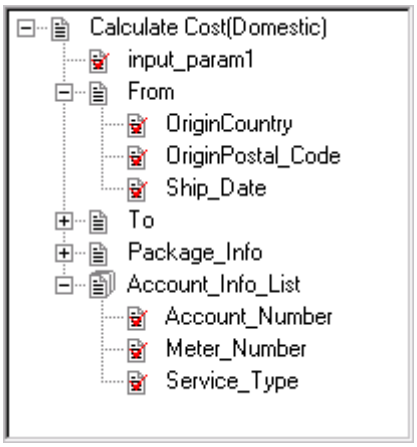
      &retinput = &CalcCostIn.AddNextDoc();

   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

GetInputDocs, AddDoc, AddValue

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

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);

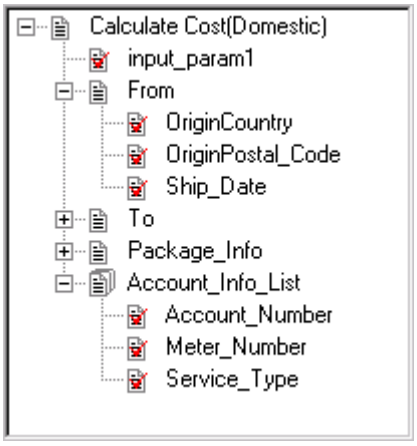
/* Get some values for input, such as &ORIGIN (code not shown) */

&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

GetInputDocs, AddDoc, AddNextDoc

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

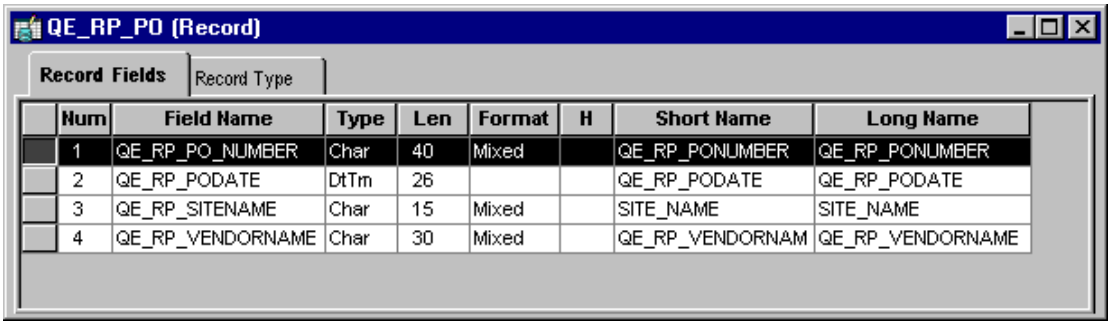
RECORD.*inputrecname*	Specify a record (SQL table) that contains the data to populate the input buffer of the Business Interlink.
RECORD.*outputrecname*	Specify a record (SQL table) that will hold the data that populates the output buffer of the Business Interlink. This is optional; it is often used 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

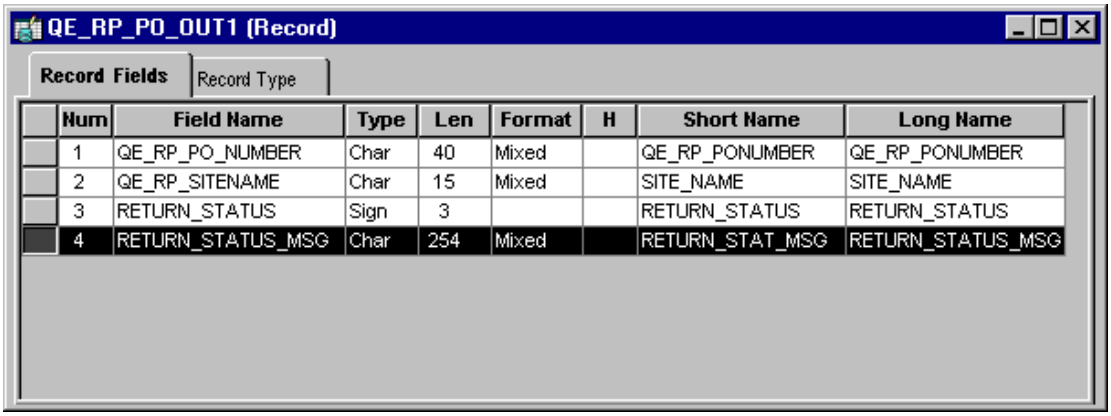
Number	Meaning
1	The Business Interlink object executes successfully
2	The Business Interlink object failed to execute
3	Transaction failed
4	Query failed
5	Missing criteria
6	Input mismatch
7	Output mismatch
8	No response from server
9	Missing parameter
10	Invalid username
11	Invalid password
12	Invalid 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.

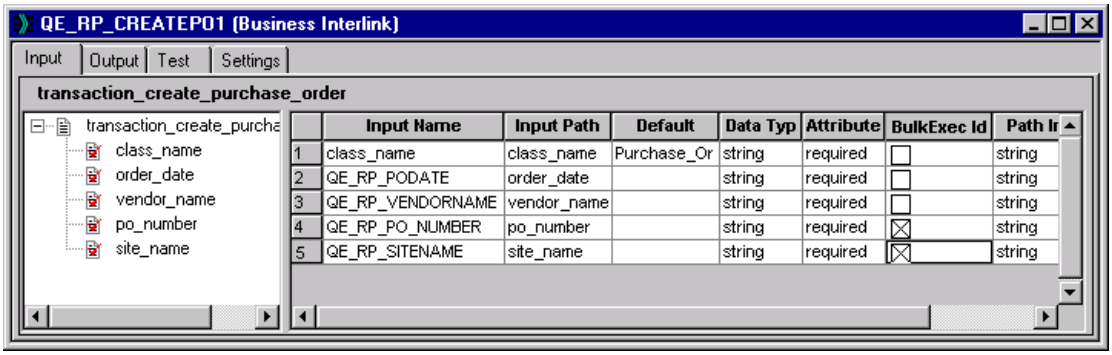


Example input record for BulkExecute

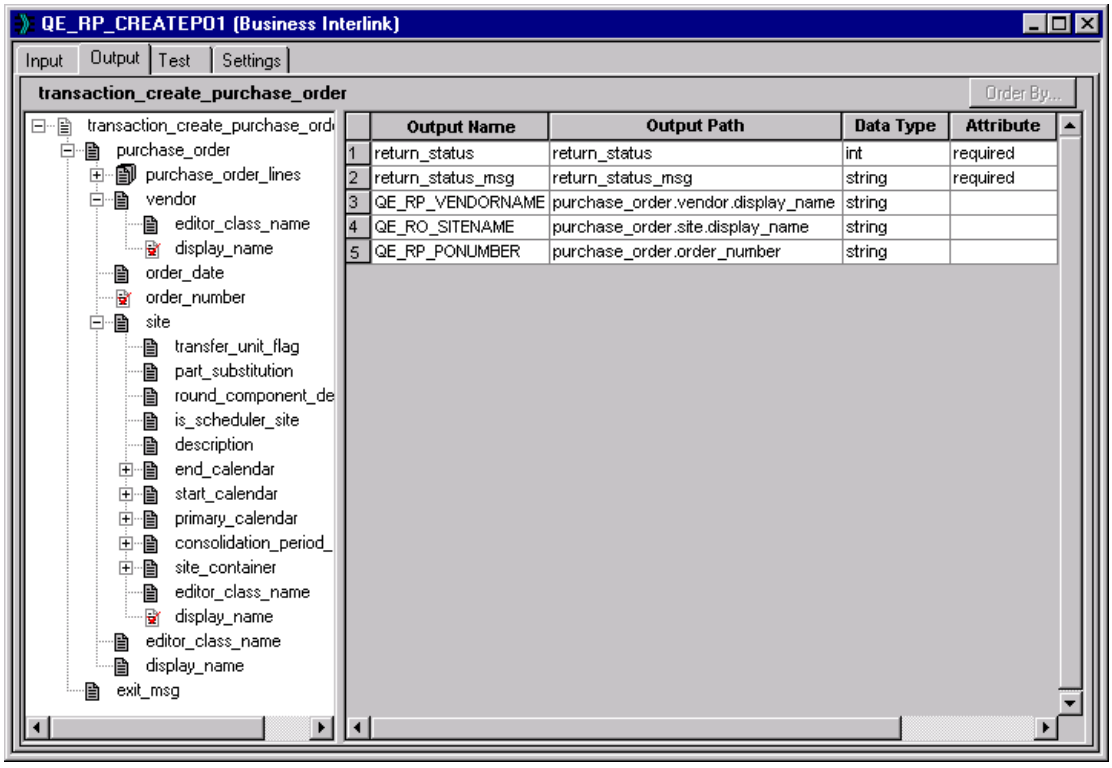


Example output record for BulkExecute

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 outputs 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

StopAtError

PeopleSoft Business Interlink Application Developer Guide

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

Execute, BulkExecute

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

Number	Meaning
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

BulkExecute, Clear

"PeopleTools Built-In Functions," GetInterlink

PeopleTools Business Interlink Application Developer Guide PeopleBook, "Generating the PeopleCode Template"

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

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

Number	Meaning
7	BulkInsertStopFailed
8	CouldNotCreateSelectCursor
9	CouldNotCreateInsertCursor
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

BulkExecute

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

&Rowset	Specify rowset object that you want to populate with data from the output buffers. This must be an existing, instantiated rowset.

Returns

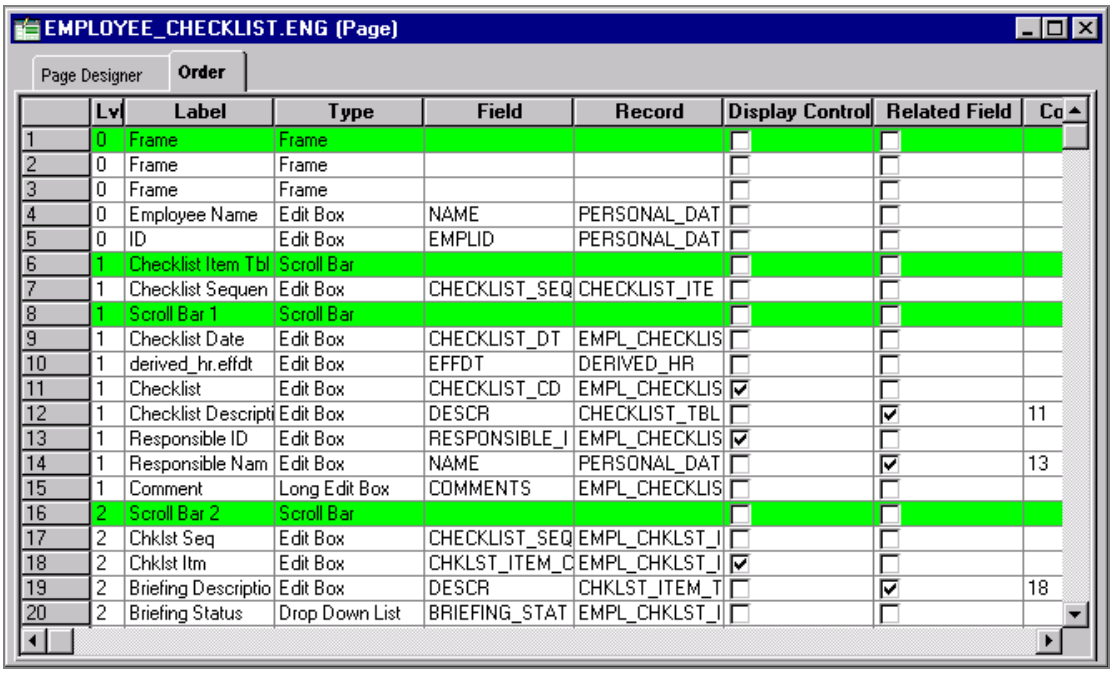
Number of rows fetched if one or more non-empty rows are returned.

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
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 Field	Co
1	0	Frame	Frame			☐	☐	
2	0	Frame	Frame			☐	☐	
3	0	Frame	Frame			☐	☐	
4	0	Employee Name	Edit Box	NAME	PERSONAL_DAT	☐	☐	
5	0	ID	Edit Box	EMPLID	PERSONAL_DAT	☐	☐	
6	1	Checklist Item Tbl	Scroll Bar			☐	☐	
7	1	Checklist Sequen	Edit Box	CHECKLIST_SEQ	CHECKLIST_ITE	☐	☐	
8	1	Scroll Bar 1	Scroll Bar			☐	☐	
9	1	Checklist Date	Edit Box	CHECKLIST_DT	EMPL_CHECKLIS	☐	☐	
10	1	derived_hr.effdt	Edit Box	EFFDT	DERIVED_HR	☐	☐	
11	1	Checklist	Edit Box	CHECKLIST_CD	EMPL_CHECKLIS	☑	☐	
12	1	Checklist Descripti	Edit Box	DESCR	CHECKLIST_TBL	☐	☑	11
13	1	Responsible ID	Edit Box	RESPONSIBLE_I	EMPL_CHECKLIS	☑	☐	
14	1	Responsible Nam	Edit Box	NAME	PERSONAL_DAT	☐	☑	13
15	1	Comment	Long Edit Box	COMMENTS	EMPL_CHECKLIS	☐	☐	
16	2	Scroll Bar 2	Scroll Bar			☐	☐	
17	2	Chklst Seq	Edit Box	CHECKLIST_SEQ	EMPL_CHKLST_I	☐	☐	
18	2	Chklst Itm	Edit Box	CHKLST_ITEM_C	EMPL_CHKLST_I	☑	☐	
19	2	Briefing Descriptio	Edit Box	DESCR	CHKLST_ITEM_T	☐	☑	18
20	2	Briefing Status	Drop Down List	BRIEFING_STAT	EMPL_CHKLST_I	☐	☐	

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

InputRowset

"Rowset Class"

PeopleTools 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

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

Execute

"GetInterlink"

PeopleTools Business Interlink Application Developer Guide PeopleBook, "Generating the PeopleCode Template"

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

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)

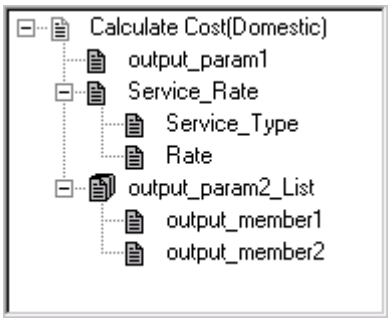
&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

/* 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

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.

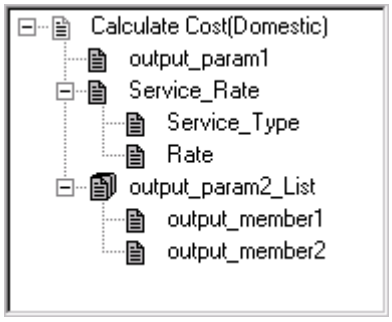
```
Local Interlink &QE_FEDEX_COST;
```

```
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)
```

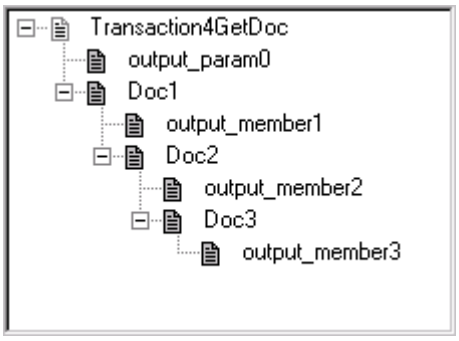
```
&CalcCostOut = &QE_4GETDOC.GetOutputDocs("");
&Docs3 = &CalcCostOut.GetDoc("Doc1.Doc2.Doc3");
&ret = &Docs3.GetValue("output_member3", &VALUE);
```

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

GetFieldType, GetFieldValue, MoveFirst, MoveNext

Using Hierarchical Data (BIDocs)

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

GetFieldCount, GetFieldValue, MoveFirst, MoveNext

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

GetFieldCount, GetFieldType, MoveFirst, MoveNext

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

Using Hierarchical Data (BIDocs)

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.

Number	Enum value and Meaning
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)

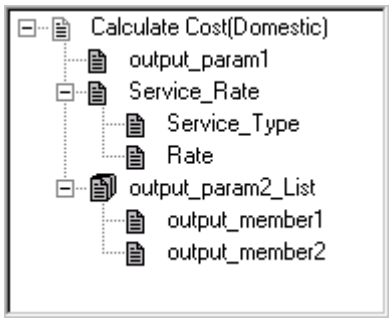
&CalcCostOut = &QE_FEDEX_COST.GetOutputDocs("");

&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

Using Hierarchical Data (BIDocs)

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.

Number	Enum value and Meaning
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;

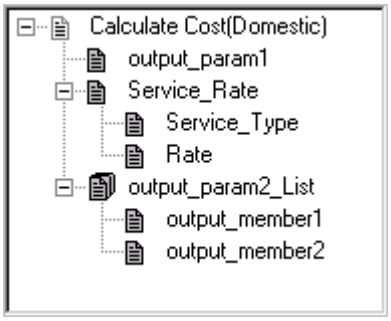
&QE_FEDEX_COST = GetInterlink(Interlink.QE_FEDEX_COST_EX);
// Get inputs, execute. (code not shown)

&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.

Return Value

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.

Number	Enum value and Meaning
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

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);

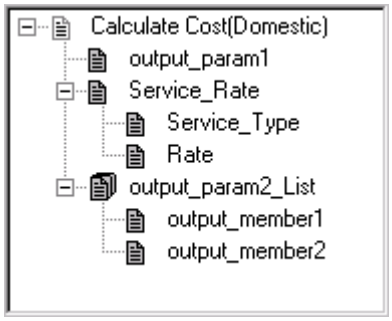
// Get inputs, execute. (code not shown)

&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

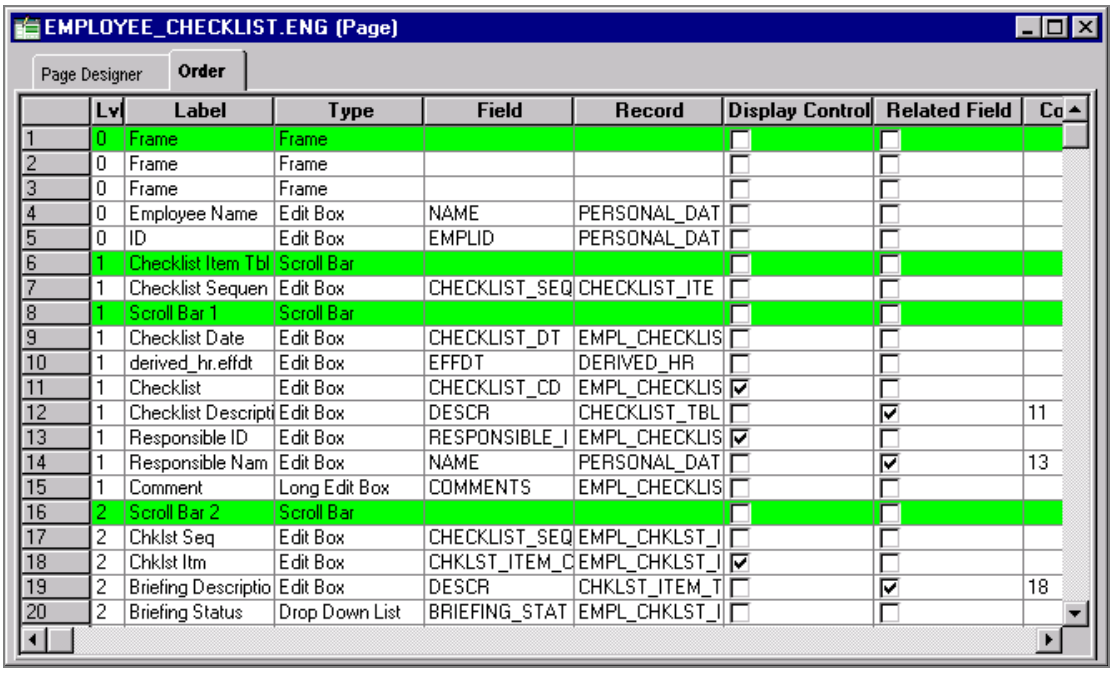
&Rowset	Specify an existing, instantiated rowset object.

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.


EMPLOYEE_CHECKLIST.ENG (Page)

Page Designer | Order

	Lvl	Label	Type	Field	Record	Display Control	Related Field	Co
1	0	Frame	Frame			☐	☐	
2	0	Frame	Frame			☐	☐	
3	0	Frame	Frame			☐	☐	
4	0	Employee Name	Edit Box	NAME	PERSONAL_DAT	☐	☐	
5	0	ID	Edit Box	EMPLID	PERSONAL_DAT	☐	☐	
6	1	Checklist Item Tbl	Scroll Bar			☐	☐	
7	1	Checklist Sequen	Edit Box	CHECKLIST_SEQ	CHECKLIST_ITE	☐	☐	
8	1	Scroll Bar 1	Scroll Bar			☐	☐	
9	1	Checklist Date	Edit Box	CHECKLIST_DT	EMPL_CHECKLIS	☐	☐	
10	1	derived_hr.effdt	Edit Box	EFFDT	DERIVED_HR	☐	☐	
11	1	Checklist	Edit Box	CHECKLIST_CD	EMPL_CHECKLIS	☑	☐	
12	1	Checklist Descripti	Edit Box	DESCR	CHECKLIST_TBL	☐	☑	11
13	1	Responsible ID	Edit Box	RESPONSIBLE_I	EMPL_CHECKLIS	☑	☐	
14	1	Responsible Nam	Edit Box	NAME	PERSONAL_DAT	☐	☑	13
15	1	Comment	Long Edit Box	COMMENTS	EMPL_CHECKLIS	☐	☐	
16	2	Scroll Bar 2	Scroll Bar			☐	☐	
17	2	Chklst Seq	Edit Box	CHECKLIST_SEQ	EMPL_CHKLST_I	☐	☐	
18	2	Chklst Itm	Edit Box	CHKLST_ITEM_C	EMPL_CHKLST_I	☑	☐	
19	2	Briefing Descriptio	Edit Box	DESCR	CHKLST_ITEM_T	☐	☑	18
20	2	Briefing Status	Drop Down List	BRIEFING_STAT	EMPL_CHKLST_I	☐	☐	

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

FetchIntoRowset

"Rowset Class**"**

***PeopleTools PeopleCode Developer's Guide PeopleBook*, "**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

GetFieldCount, GetFieldType, GetFieldValue, MoveNext

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

GetFieldCount, GetFieldType, GetFieldValue, MoveFirst

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

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.

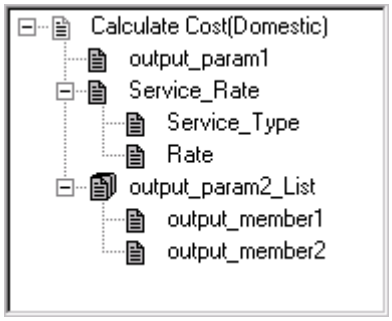
```
Local Interlink &QE_FEDEX_COST;
Local number &count;
Local BIDocs &CalcCostOut;
Local BIDocs &OutlistDoc;
Local number &ret;

&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();
```

Business Interlink BIDocs Methods

This section describes the PeopleCode methods you use with Incoming Business Interlinks.

See Also

PeopleSoft Business Interlink Application Developer Guide

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

attributename	String. The name of the attribute.
attributevalue	String. The value of the attribute.

Return Value

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

GetStatus

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

comment	String. The comment.

Return Value

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

GetStatus

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

instruction	String. The processing instruction.

Return Value

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

GetStatus

AddText

Syntax

AddText(*text*)

Description

The **AddText** method adds text to an XML element referenced by a BiDocs object.

Parameters

text String. The text.

Return Value

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

GetStatus

CreateElement

Syntax

CreateElement(*elementname*)

Description

The **CreateElement** method creates an XML element with the given name within a BiDoc object.

Parameters

elementname String. The XML element name.

Return Value

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.

Return Value

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

attributenumber Number. The index number of the attribute.

Return Value

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

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.)

Return Value

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

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.)

Return Value

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

XMLstring	A string containing an XML document.

Return Values

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

GetStatus

Business Interlink BIDocs Properties

This section describes the PeopleCode properties you use with Incoming Business Interlinks.

See Also

PeopleSoft Business Interlink Application Developer Guide

AttributeCount

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

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

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

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

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

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;
```

See Also

PeopleTools Business Interlink Application Developer Guide PeopleBook, "Generating the PeopleCode Template"

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 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.

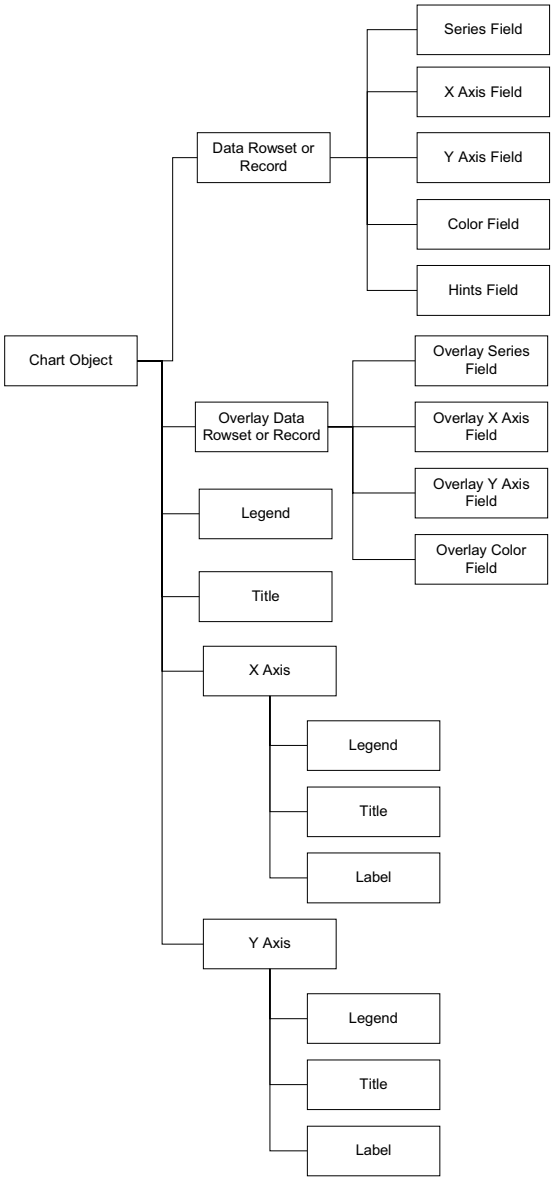
There is only one object used with the chart class, that is, a chart object. There are no subobjects. All methods and properties are used with the chart object. You do not have to instantiate a subobject to access its properties and methods.

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.

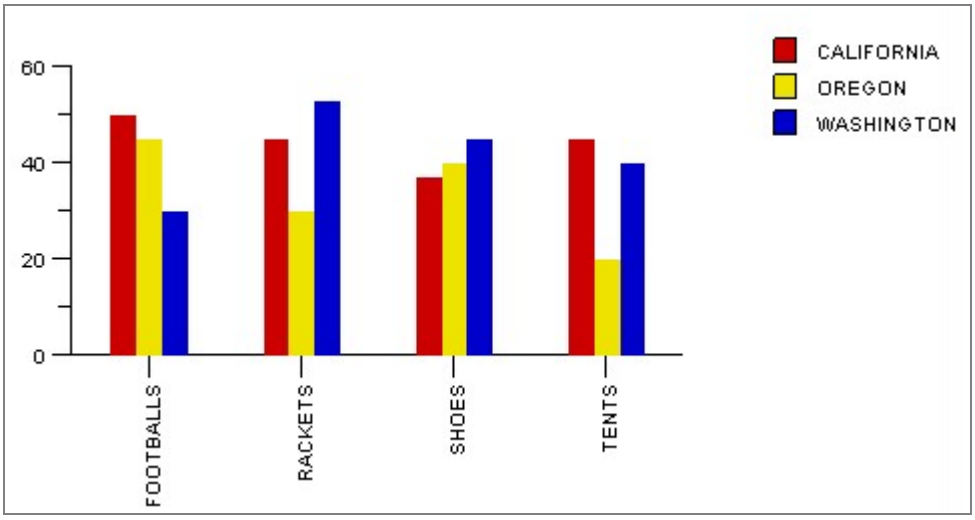
Overlay	Related data represented by a line drawn over the background chart. You can also specify a data series, with different line types for each overlay.
Legend	Text that identifies the different series in the chart.
Title	There are three titles: Main, X axis, and Y axis. These are text that identifies 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:

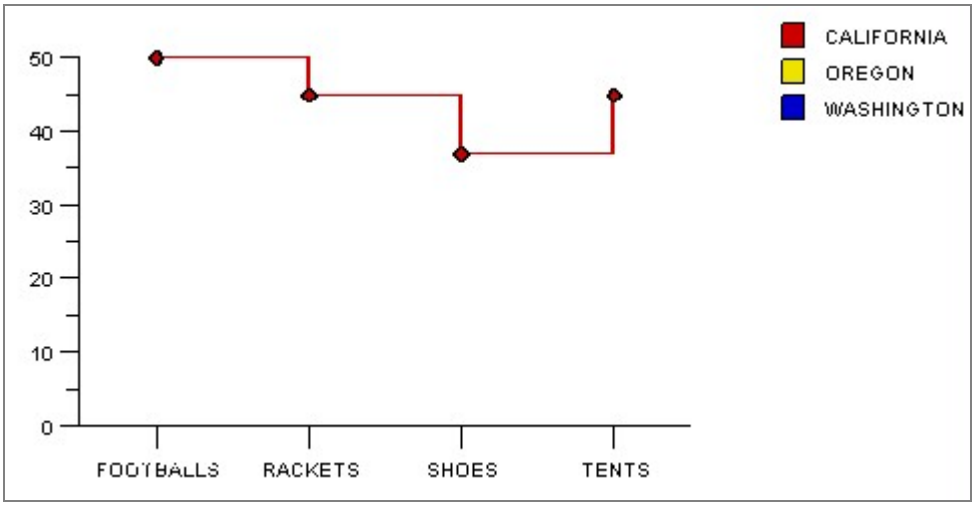
- 2D Bar
- 2D Histogram
- 2D Horizontal Bar
- 2D Horizontal Percent Bar
- 2D Horizontal Stacked Bar
- 2D Line
- 2D Percent Bar
- 2D Pie
- 2D Stacked Bar
- 3D Bar
- 3D Percent Bar
- 3D Pie
- 3D Stacked Bar

2D Bar



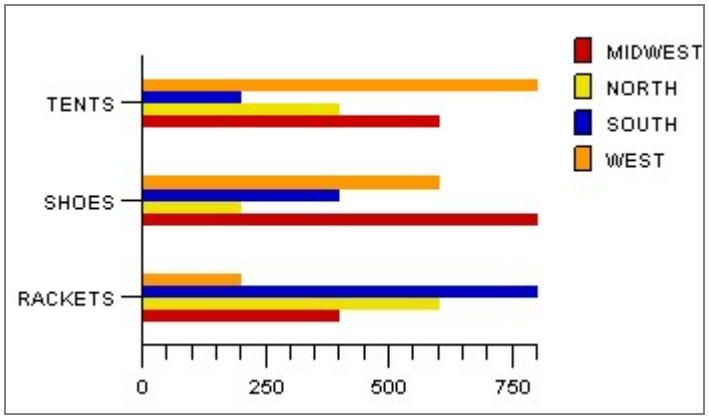
Type=%ChartType_2DBar

2D Histogram



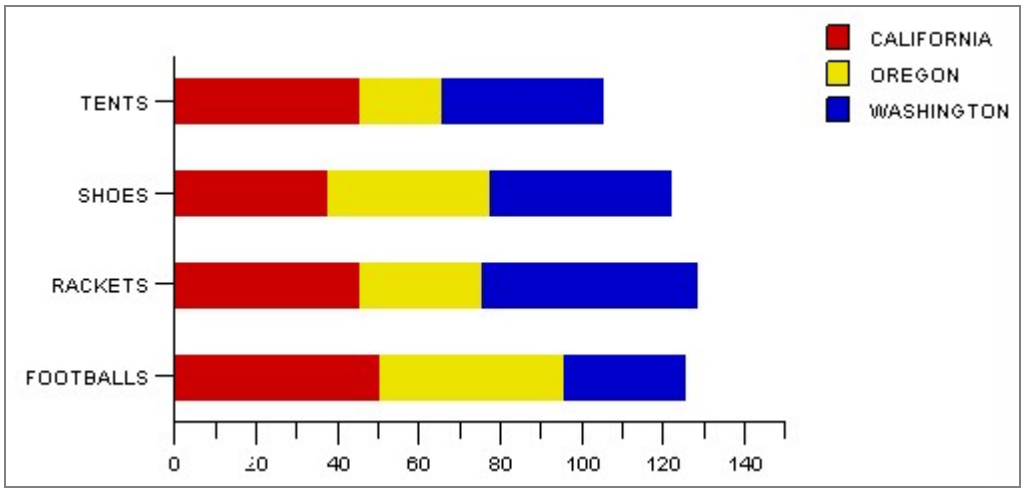
Type = %ChartType_2DHistogram

2D Horizontal Bar



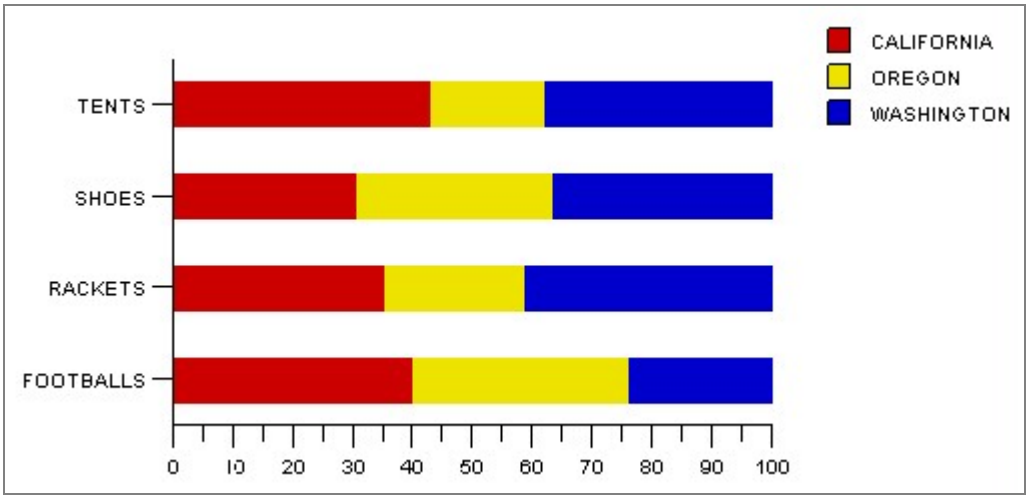
Type = %ChartType_2DHorizBar

2D Horizontal Stacked Bar



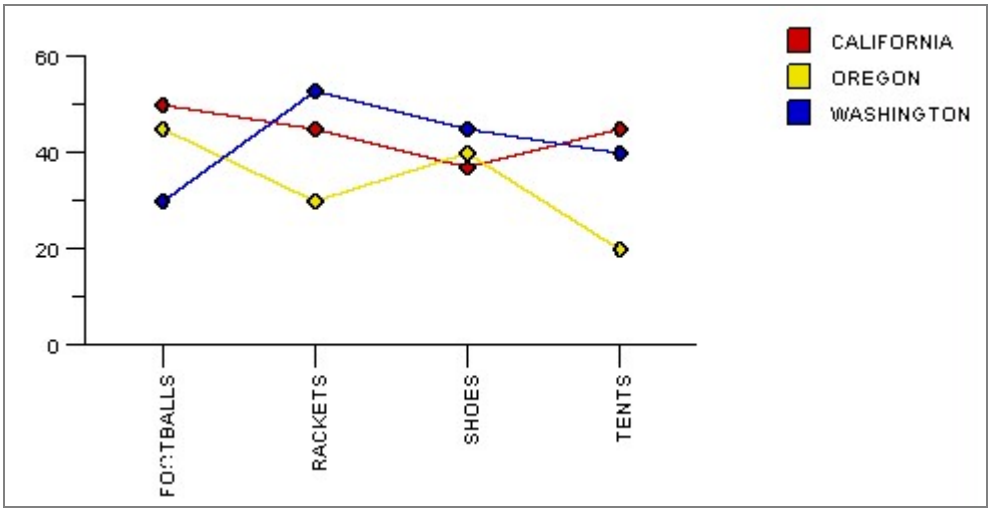
Type = %ChartType_2DHorizStackedBar

2D Horizontal Percent Bar



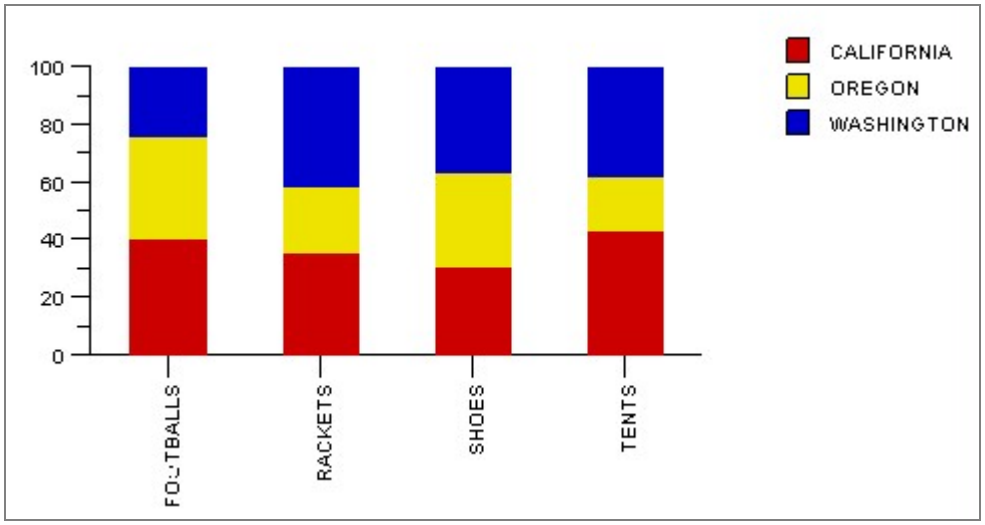
Type = %ChartType_2DHorizPercentBar

2D Line



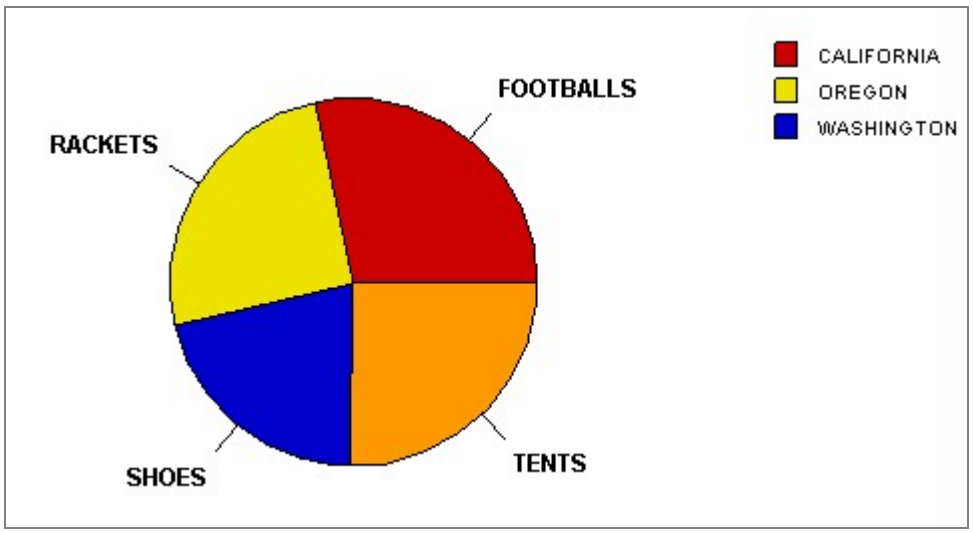
Type = %ChartType_2DLine

2D Percent Bar



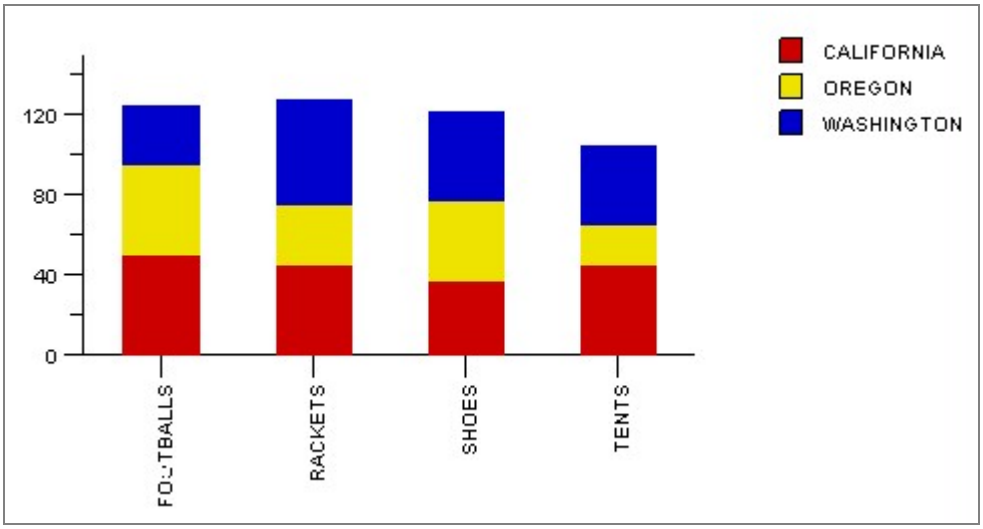
Type = %ChartType_2DPercentBar

2D Pie



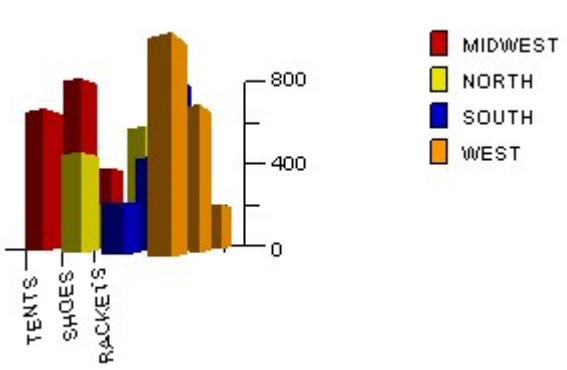
Type = %ChartType_2DPie

2D Stacked Bar



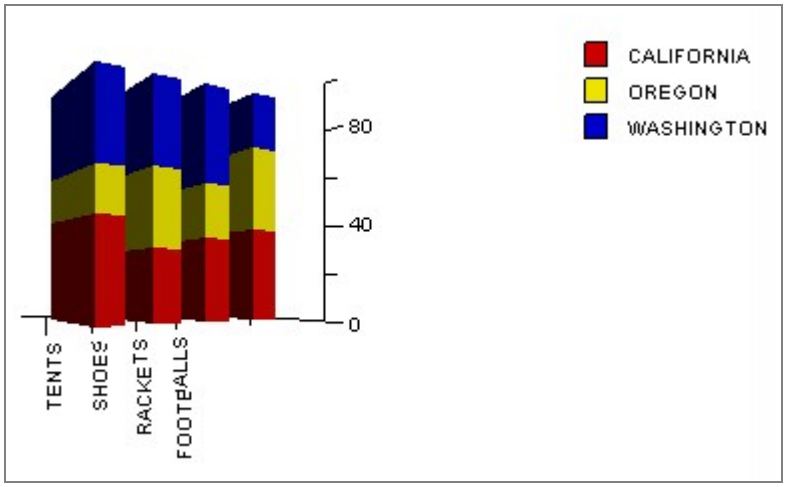
Type = %ChartType_2DStackedBar

3D Bar



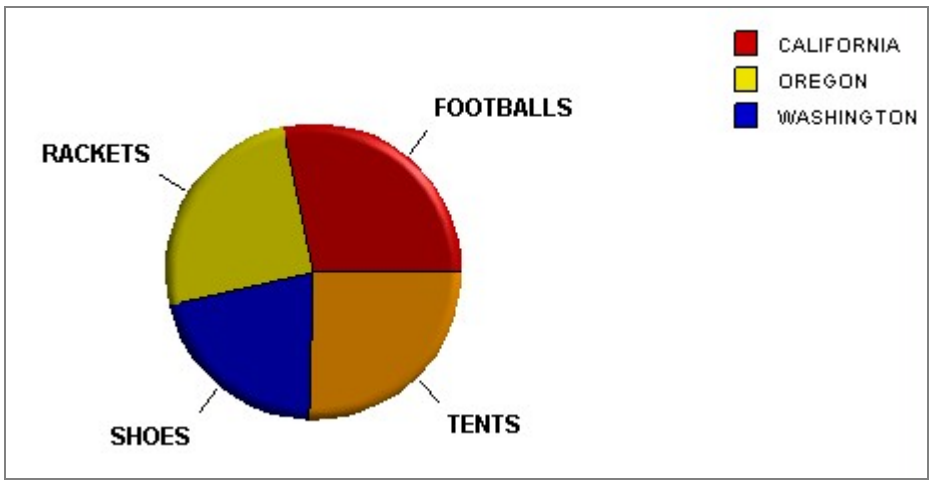
Type = %ChartType_3DBar

3D Percent Bar



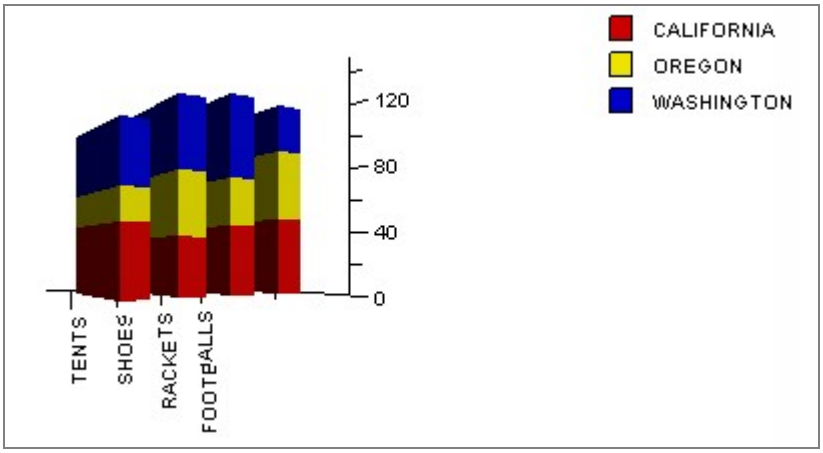
Type = %ChartType_3DPercentBar

3D Pie



Type = %ChartType_3DPie

3D Stacked Bar



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

PeopleTools PeopleBook: 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

PeopleTools PeopleBook: 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

Creating a Chart Using an iScript

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 Built-in Functions

CreateObject

GetChart

GetChartURL

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 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

DataStartRow, DataWidth, SetDataXAxis, SetDataYAxis

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

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

SetDataXAxis, SetDataYAxis

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

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

SetDataXAxis, SetDataYAxis

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

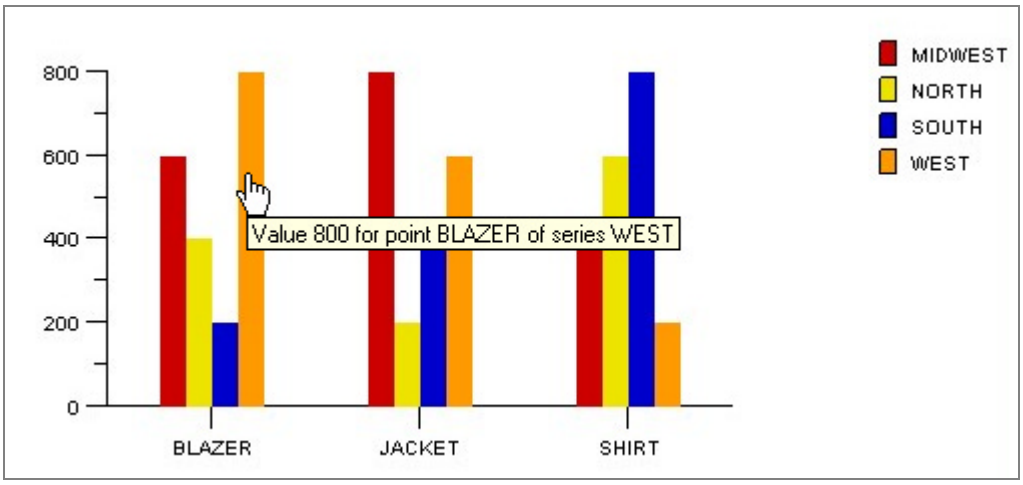
If you do *not* specify data hints, but you *do* specify both the data series and data methods, the hints appear as follows:

```
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

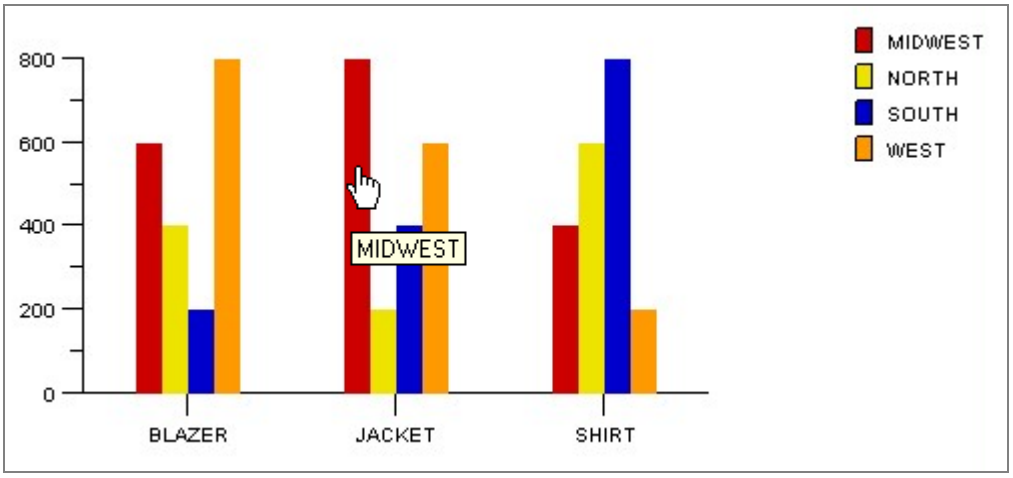
The following is an example of this type of data hints. The pop-up menu appears when the cursor hovers over any data value.



Data hints with multiple series example

Data Hints, Data, and Series

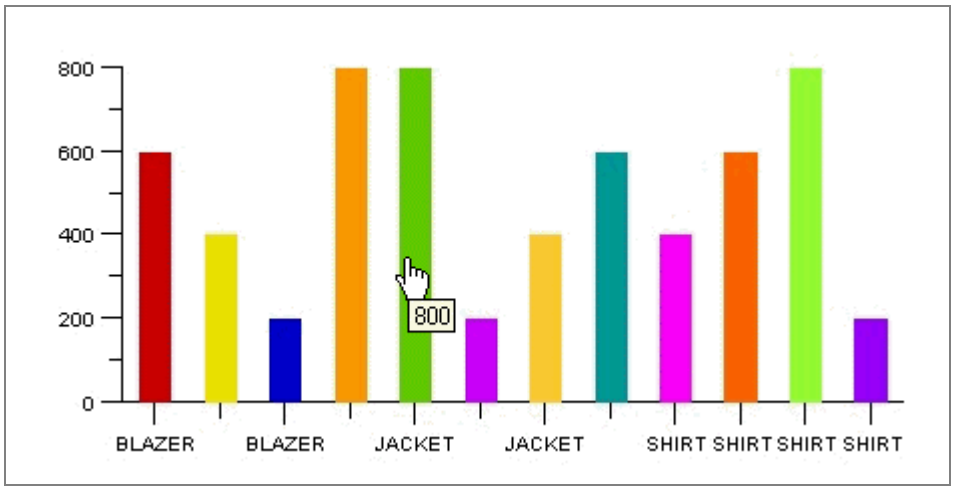
If you specify data hints, that is the value that appears. In the following example, the data hints are set to the same value as the series is set, that is, the region.



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

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

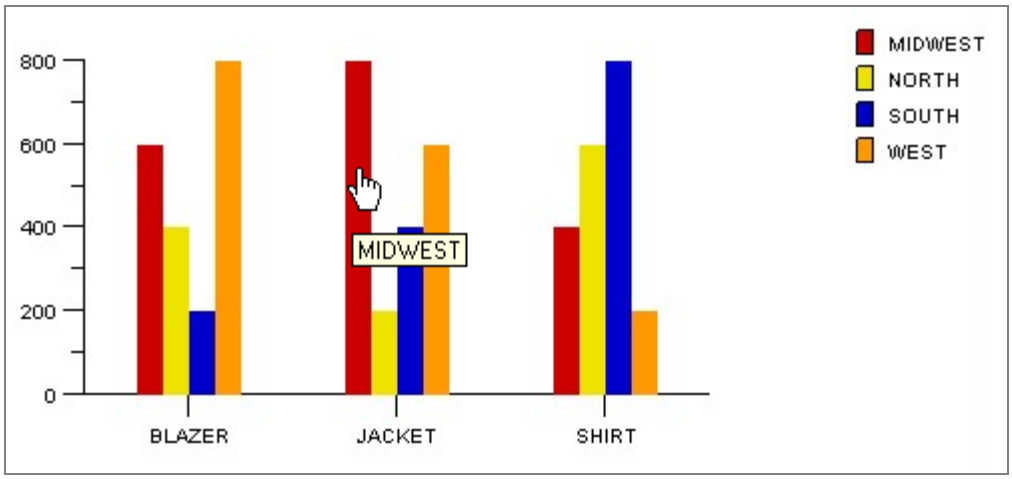
None.

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

SetDataXAxis, SetDataSeries

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. 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

FieldName	Specify a field name that's used to populate the URL for each data point.

Returns

None.

Example

```
&MyChart.SetDataURLs("http://pdtn089/HRKL/scripts/popup.html");
```

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

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

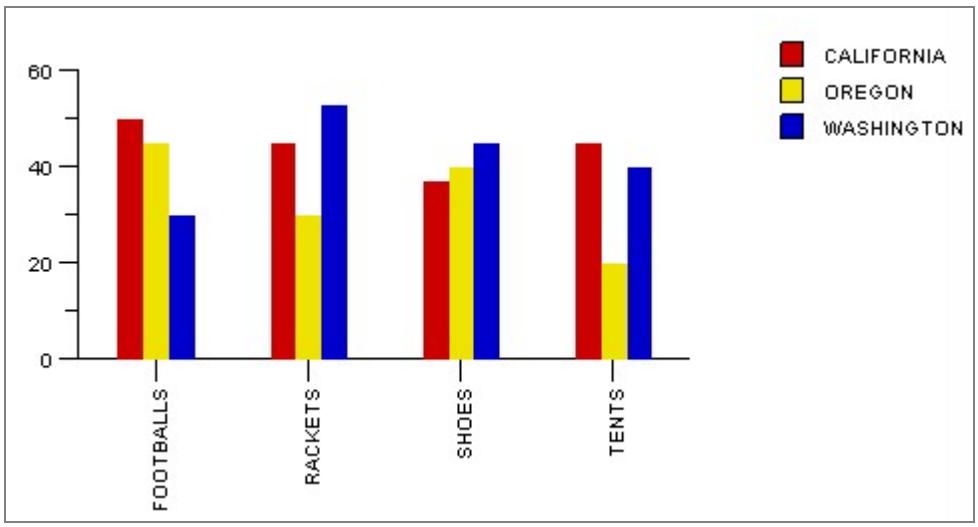
None.

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

SetDataYAxis, SetData

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 under the last negative number, not at 0.

Parameters

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

SetDataXAxis, SetData

Chart Data Properties

These properties are used to set X and Y axes data.

DataStartRow

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

DataWidth

DataWidth

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

DataStartRow

ImageMap

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

Creating a Chart Using an iScript

Chart Appearance Properties

These properties are used to set the type of graph, the line style, and so on.

GridLines

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*
1	%ChartGrid_None	Don't show grid lines (default)
2	%ChartGrid_Horizontal	Show horizontal grid lines
3	%ChartGrid_Vertical	Show vertical grid lines
4	%ChartGrid_Both	Show both vertical and horizontal grid lines

This property is read-write.

Height

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

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.

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. **Note.** Setting this property has no effect if the chart contains overlay data.

The default value for this property is False.

This property is read-write.

See Also

IsPlainImage

IsPlainImage

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

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

OLLineType

RotationAngle

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

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

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

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

Chart Types

Width

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 **SetDataColorArray** method.
0	%ChartColor_Black	Black
1	%ChartColor_Blue	Blue
2	%ChartColor_Cyan	Cyan
3	%ChartColor_DarkGray	Dark Gray
4	%ChartColor_Gray	Gray
5	%ChartColor_Green	Green
6	%ChartColor_LightGray	Light Gray
7	%ChartColor_Magenta	Magenta
8	%ChartColor_Orange	Orange
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

Parameters

&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 Chart Colors for a list of the color values.

Returns

None.

Example

The following example sets the four series in the chart:

```
&NumArray = CreateArray(1, 3, 6, 9);

&oChart.SetColorArray(&NumArray);
```

See Also

"Array Class"

SetDataColor

Syntax

```
SetDataColor(Record_Name.Field_Name)
```

Description

Use the **SetDataColor** method to specify a field that contains the color for each data point.

Note. This method has no effect when used with an overlay.

Parameters

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 Chart Colors for a list of the color values.

Returns

None.

SetOLDataYAxis

Syntax

```
SetOLDataYAxis(Record_Name.Field_Name)
```

Description

Use the **SetOLDataYAxis** method to set the color for each data point of the overlay.

Record_Name.Field_Name	Specify the name of the record, and the field on that record, that contains the colors for each overlay data point. You can specify either a constant or a number.

See Chart Colors for a list of the color values.

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.

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

&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

SetDataXAxis

"Array Class"

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.

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

&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

SetDataXAxis, YAxisTicks, YAxisMax

"Array Class"

Chart Axes Properties

These properties are used to further control the appearance of the axes.

XAxisLabelOrient

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

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

Style Sheets and Charts

YAxisLabelOrient

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

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

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

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

Style Sheets and Charts

YAxisTicks

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

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

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:

Number 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

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

Style Sheets and Charts

XAxisTitle

Use this property to specify the text for the X axis title.

This property is read-write.

XAxisTitleOrient

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:

Number 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

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

Style Sheets and Charts

YAxisTitle

Use this property to specify text of the title for the Y axis.

This property is read-write.

YAxisTitleOrient

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:

Number 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

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

Style Sheets and Charts

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

&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

SetDataSeries

"Array Class"

Chart Legend Properties

These properties are used for each series legend.

HasLegend

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

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

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

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

Style Sheets and Charts

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

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

SetDataXAxis, SetDataYAxis, SetData

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

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

SetDataSeries

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

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

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

SetDataXAxis, SetDataYAxis

Chart Overlay Properties

These properties are used to set and manipulate overlay data.

OLLineType

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

LineType

OLType

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
1	%ChartType_2DLine	Overlay is drawn as a 2D line.
2	%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);

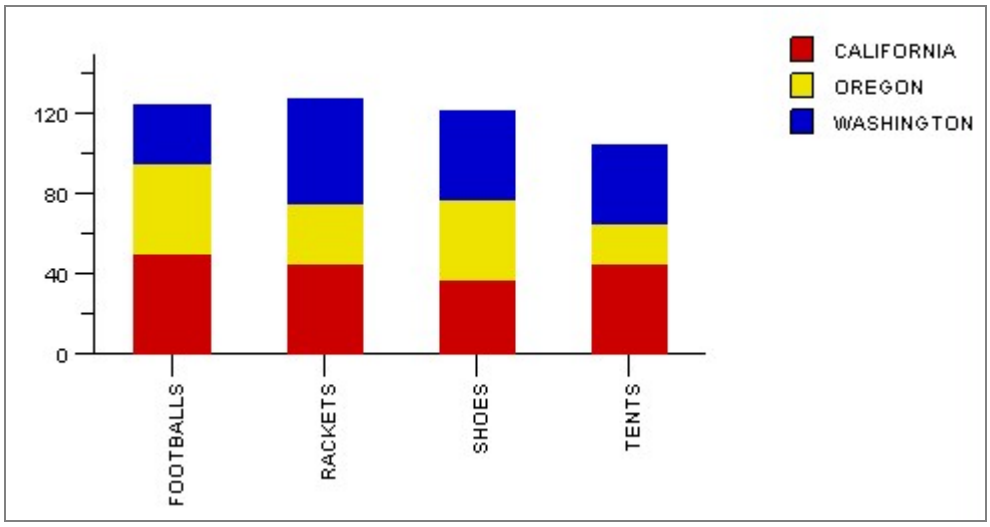
&oChart.SetDataSeries(QE_CHART_RECORD.QE_CHART_REGION);

&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);

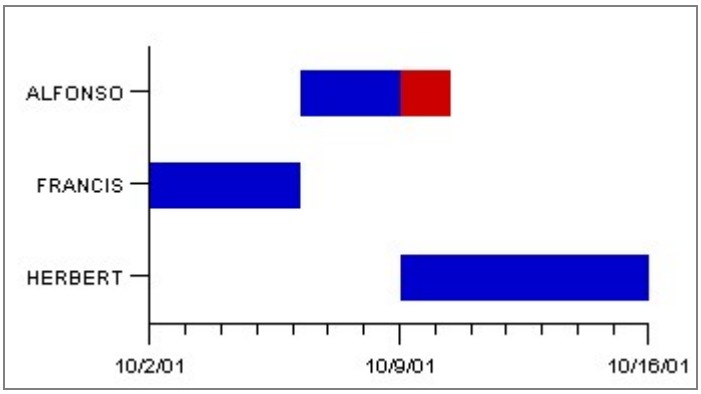
&oChart.SetDataXAxis(QE_CHART_RECORD.QE_CHART_EMPLOYEE);

/* 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 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

Reusing Existing Code

"Session Class"

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:

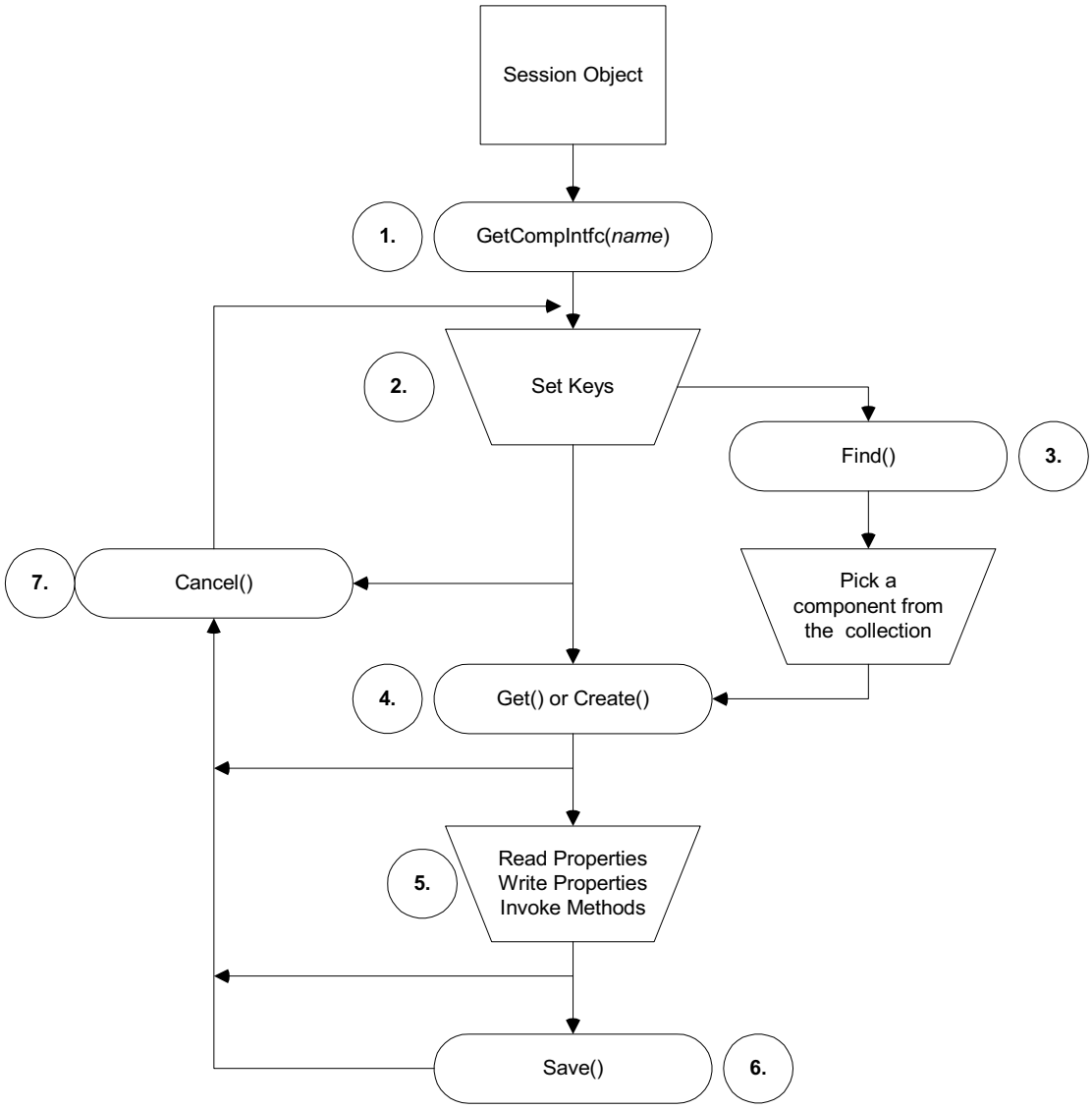
    - Execute the Find standard method on the Component Interface. This returns a collection of Component Interfaces with their key values filled out. The user can then select the unique Component Interface they want.

    - 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. **Note.** 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

Component Interface Class Methods

PeopleSoft Component Interfaces

PeopleTools PeopleBook, 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:
  - EditHistoryItems
  - GetHistoryItems
  - InteractiveMode

  These properties must be set before the Component Interface is populated with data (that is, before you use the **Get** or **Create** method.)

  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:
  - CreateKeyInfoCollection
  - FindKeyInfoCollection
  - GetKeyInfoCollection

- PropertyInfoCollection

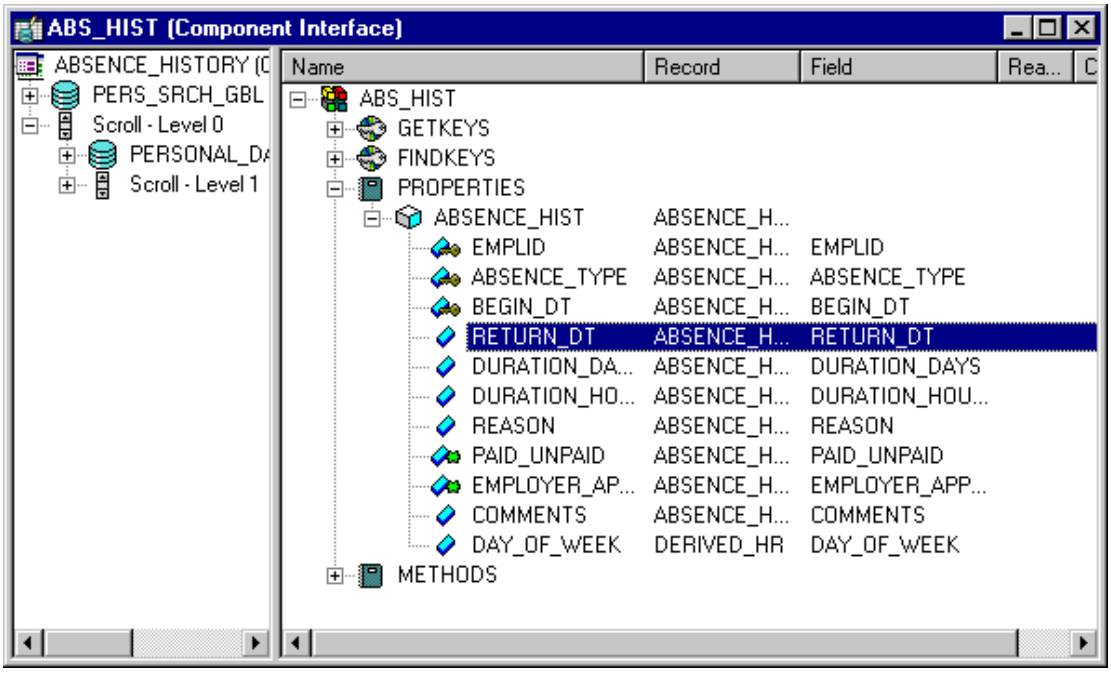
See Also

Component Interface Class Properties

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;
```

Declaring 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

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

"Rowset Class", Refresh

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

Traversing a Component Interface and Using Data Collections

Working With Effective-Dated Data

Accessing the Structure of 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).[CollectionNa
me(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 Also

"Session Class," Error Handling, Source

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).[CollectionN
ame(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 Also

"Session Class," Error Handling, Source

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 Also

   Data Collection Methods

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)
```

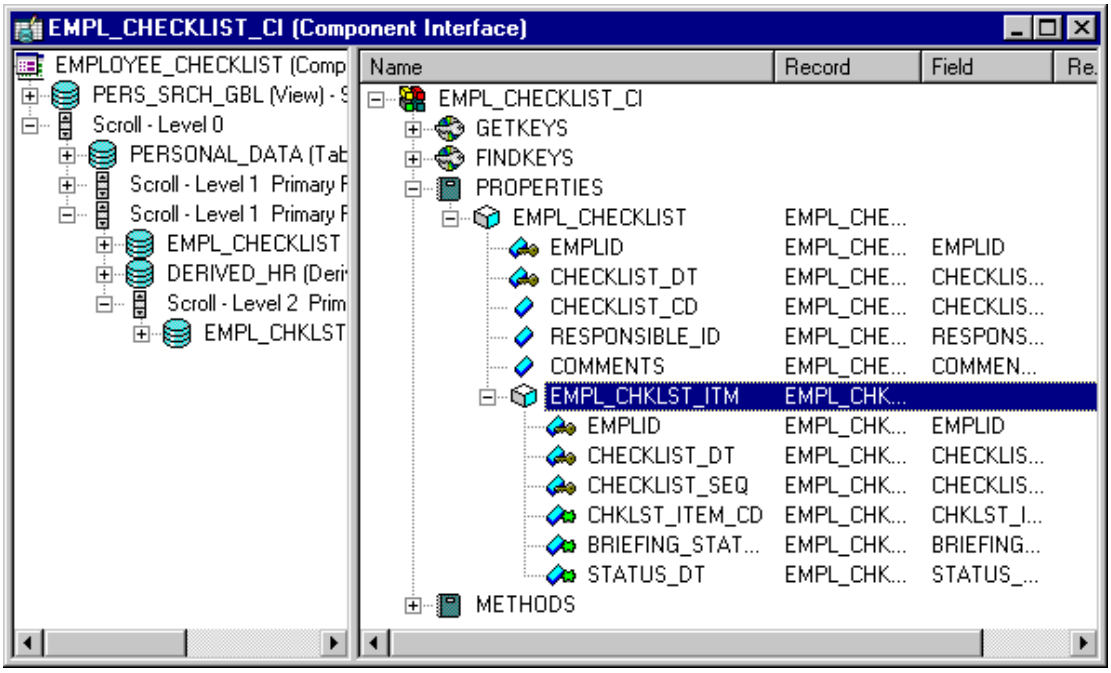
A *collection* is a set of similar things, like a group of already existing Component Interfaces. Most collections have the same standard properties and methods, with some additional ones specific to that collection. For example, all collections have the property **Count**, which indicates how many items 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

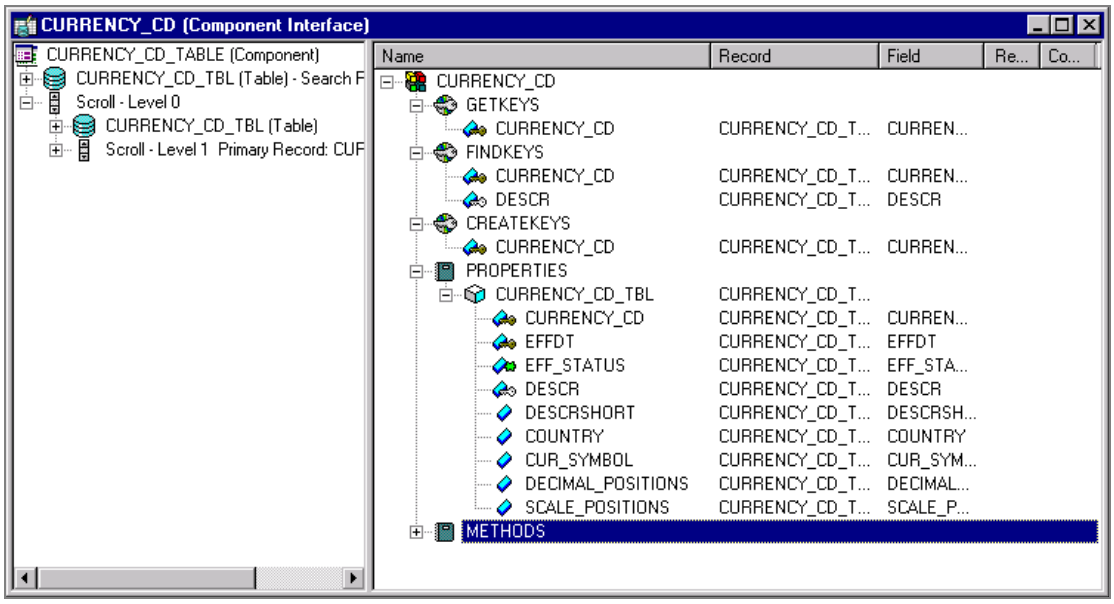
Data Collection Methods, Data Collection Properties

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Data Buffer Access"

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

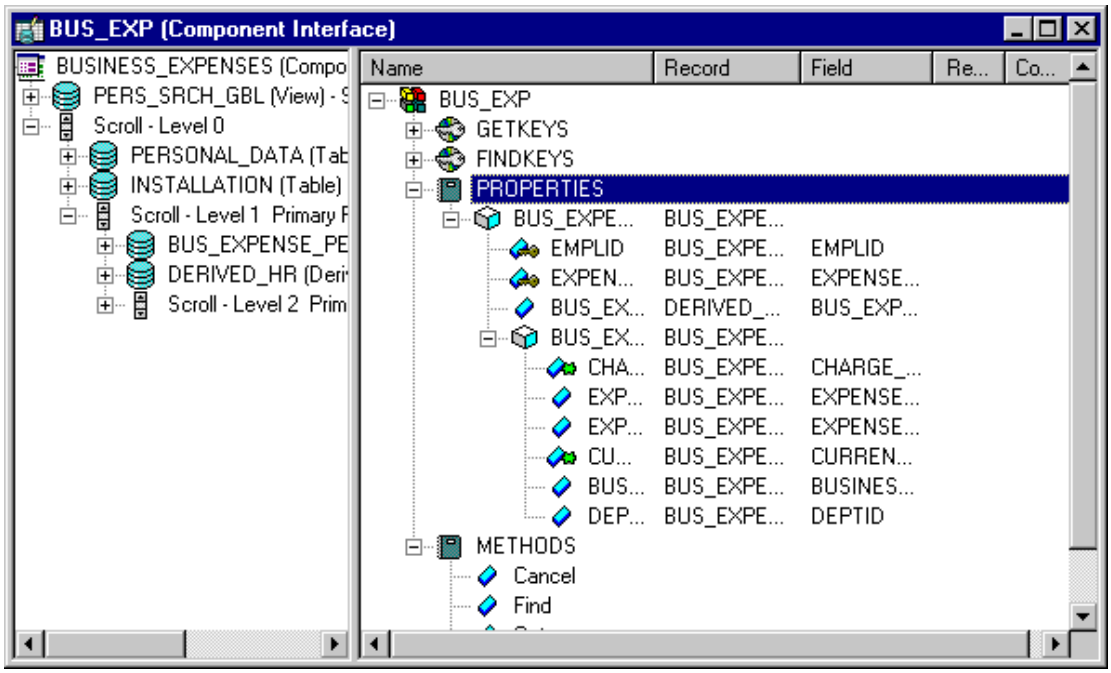
Sub MAIN()
    CURRENCY_CD
End Sub
```

Inserting or Deleting a Row of Data Example

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).[CollectionNa
me(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 Also

"Session Class," Error Handling, Source

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

CurrentItem, GetEffectiveItem, GetHistoryItems

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

PeopleTools PeopleBook: Component Interface, "Developing the Component Interface," Associating a Component Interface With a Menu

Session Object Methods

Component Interfaces don't have any built-in functions. They are instantiated from a session object.

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

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.

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.

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. **Note. 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

&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. **Note.** 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

&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

PeopleTools Integration Tools PeopleBook, "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. **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. 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

&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;
```

Related Topics

CopySetupRowsetDelta, CopyRowset

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. **Note.** 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

&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

CopySetupRowset, CopyRowsetDelta

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

The **Find** method returns a reference to a data collection matching the key values that were set prior to using the Find method. This data collection may have zero elements in it if no items matching the key values you set were found. The Find method limits the number of returned instances to 300. If more instances are found, an error is logged to the PSMessages collection.

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

Data Collection Methods

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

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

SetPropertyByName

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

InteractiveMode

"Session Class," ErrorPending

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

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

GetPropertyByName

Component Interface Class Properties

In this section, we discuss the Component Interface class properties. The properties are discussed in alphabetical order.

ComponentName

This property returns the name of the Component Interface, as defined in PeopleSoft Application Designer, as a string.

This property is read-only.

CreateKeyInfoCollection

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

Accessing the Structure of a Component Interface

EditHistoryItems

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

GetHistoryItems

FindKeyInfoCollection

This property returns a CompIntfPropInfoCollection collection that contains a CompIntfPropInfoCollection object for every key in FINDKEYS.

This property is read-only.

See Also

Accessing the Structure of a Component Interface

GetHistoryItems

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

EditHistoryItems

GetKeyInfoCollection

This property returns a CompIntfPropInfoCollection collection that contains a CompIntfPropInfoCollection object for every key in GETKEYS.

This property is read-only.

See Also

Accessing the Structure of a Component Interface

InteractiveMode

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

"Session Class," ErrorPending

PropertyInfoCollection

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

Accessing the Structure of a Component Interface

StopOnFirstError

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

Traversing a Component Interface and Using Data Collections

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.)

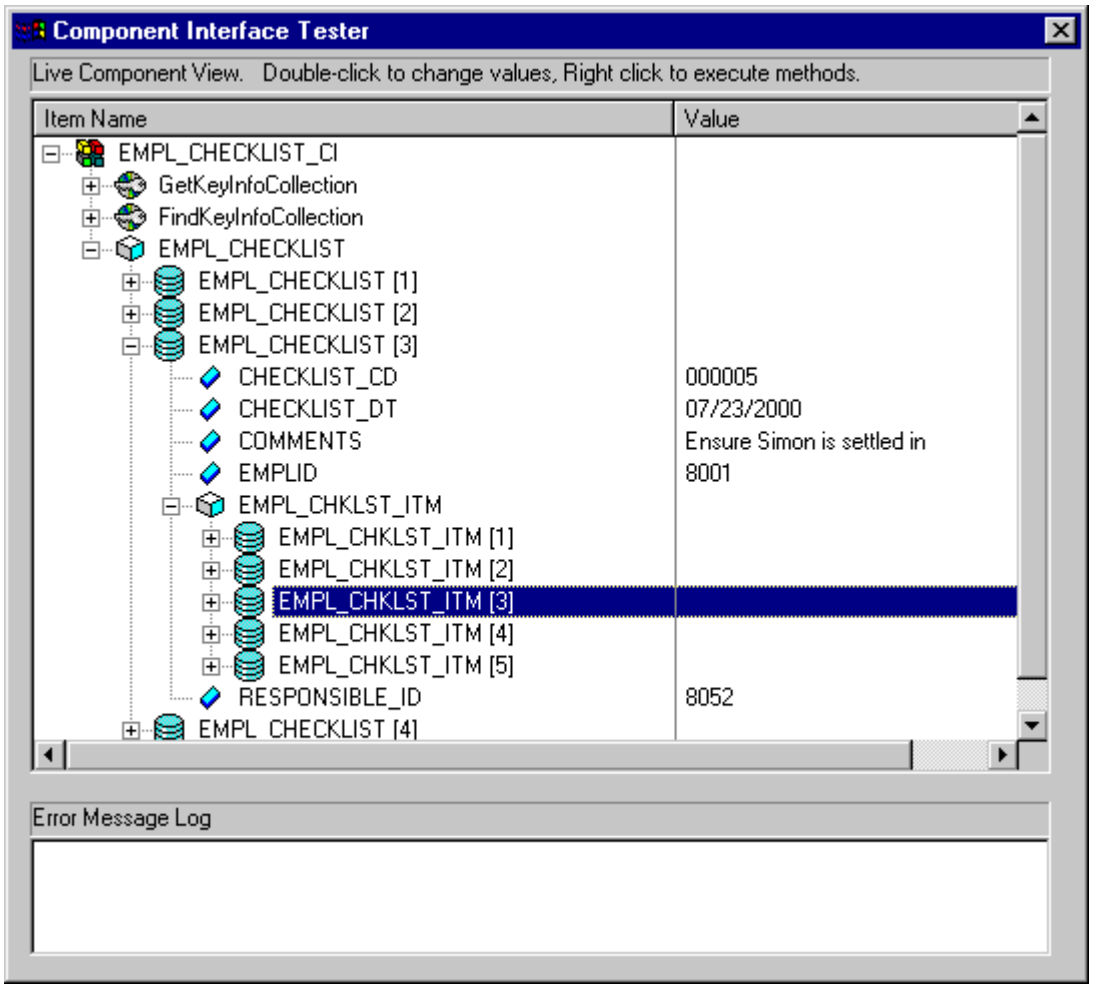
If you set the **InteractiveMode** property to True, any RowDelete PeopleCode runs immediately after you execute DeleteItem. If this property is set to False, any RowDelete PeopleCode runs after you execute the **Save** method.

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

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

"PeopleCode Built-In Functions," String

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

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

"PeopleCode Built-in Functions," String

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 Also

ItemNum

PeopleTools PeopleBook, PeopleSoft Component Interface, "Developing the Component Interface," Testing the Component Interface

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;
```

Data Collection Properties

This section explains the following Data Collection properties:

- Count
- CurrentItemNumber

Count

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

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

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 Also

GetCompIntfc

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.

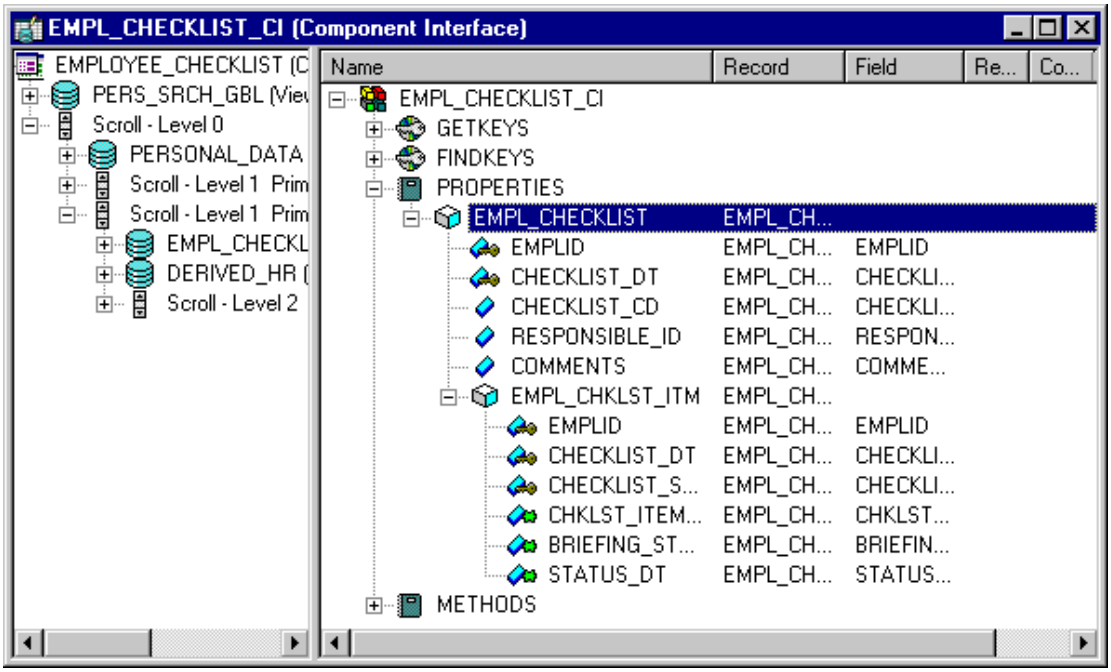
  - CREATEKEYS: A collection containing the key fields of the search record specified to be used in Add mode. Use the **CreateKeyInfoCollection** property to instantiate the CompIntfPropInfoCollection collection.
  - GETKEYS: A collection containing the primary required key fields from the primary search 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

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

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 Returned	*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

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

This property returns True if the property marked Read Only in the Component Interface Definition; False otherwise.

This property is read-only.

Key

This property returns True if the object executing the property is a key, False otherwise.

This property is read-only.

LabelLong

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

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

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

This property returns the name of the object executing the property as a string.

This property is read-only.

Prompt

This property returns True if the object executing the property is associated with a prompt table, False otherwise.

This property is read-only.

PropertyInfoCollection

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

This property returns True if the object executing the property is a required property, False otherwise.

This property is read-only.

Type

This property returns the field type, as a number, of the object. The values are:

Value Returned	*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

This property returns True if the object executing the property is associated with an XLAT table, False otherwise.

This property is read-only.

YesNo

This property returns True if the object executing the property is associated with the Yes/No table, False otherwise.

This property is read-only.

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;
```

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 Function and Language Constructs

CreateException

Throw

Try

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

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

SetSubstitution

PeopleTools PeopleBook: Data Management, "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

Throw

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

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

GetSubstitution

PeopleTools PeopleBook: Data Management, "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

Output

Exception Class Properties

In this section, we discuss the Exception class properties, in alphabetical order.

Context

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

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

This property is the message number for a message describing the exceptional condition.

This property is read-only.

MessageSetNumber

This property is the message set number for a message describing the exceptional condition.

This property is read-only.

MessageSeverity

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:

Constant Value	*String Value*	*Description*
%MsgSeverity_Cancel	C	Message severity is Cancel.
%MsgSeverity_Error	E	Message severity is Error.
%MsgSeverity_Message	M	Message severity is Message.
%MsgSeverity_Warning	W	Message severity is Warning.

This property is read-only.

SubstitutionCount

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

Field Class Overview

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

PeopleTools PeopleBook: 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

GetField

Field Class Methods

In this section, we discuss the Field class methods. The methods are discussed in alphabetical order.

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

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
Chartfield");
End-If;
```

See Also

"PeopleCode Built-in Functions," SetDBFieldAuxFlag

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

GetShortLabel, Label

"PeopleCode Built-in Functions," SetLabel

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

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

"PeopleCode Built-in Functions," GetField

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

GetLongLabel, Label

"PeopleCode Built-in Functions," SetLabel

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

SetDefault

"Record Class," SearchClear

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 method can't be used by a PeopleCode program that's been called by a Component Interface. You should put a condition around this function, testing whether there's an existing Component Interface.

```
If %CompIntfcName Then
     /* process is being called from a Component Interface */
     /* do BC specific processing */
Else
     /* do regular processing */
     .  .  .
End-if;
```

Returns

None.

Parameters

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).SetCurso
rPos(PAGE.pagename);
```

See Also

SetDefault

"PeopleCode Built-in Functions," SetCursorPos

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

"Record Class," SetDefault

PeopleTools PeopleBook: 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

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 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. 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

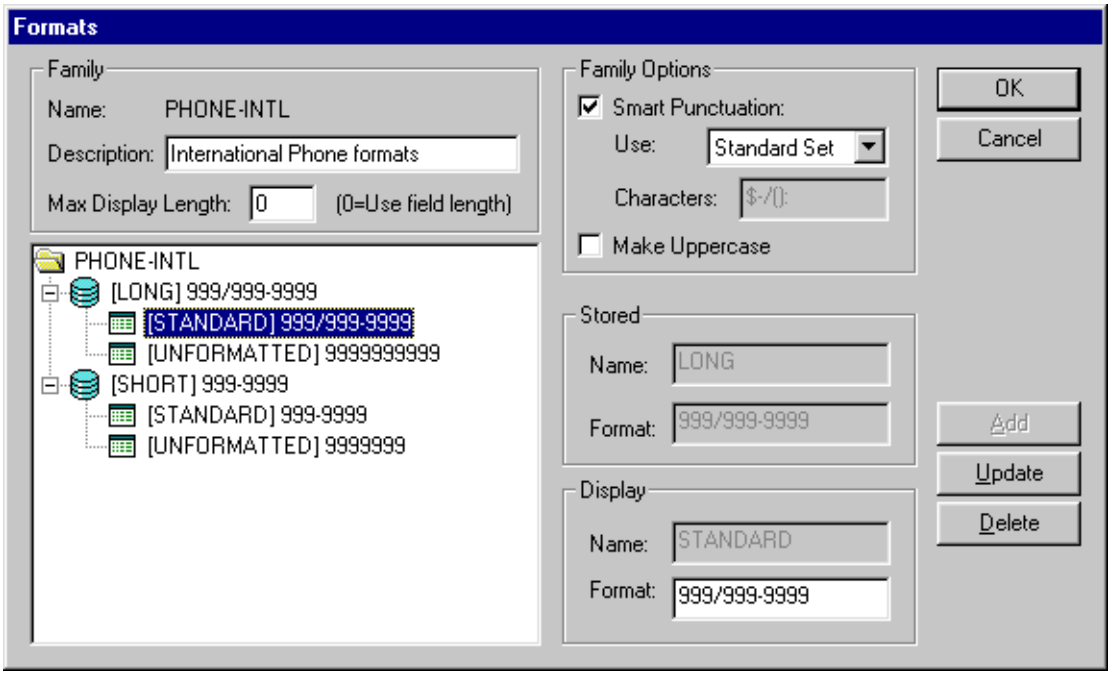
"Rowset Class," DataAreaCollapsed

DisplayFormat

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 custom 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

StoredFormat

DisplayOnly

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

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

"Record Class," ExecuteEdits

"Message Class," ExecuteEdits, MessageNumber, MessageSetNumber

Enabled

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Referencing Data in the Component Buffer," What Record Fields Are in the Component Buffer?

IsKey

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

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.

IsRequired

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

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

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

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

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

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

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. **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.

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

GetShortLabel, GetLongLabel

LabelImage

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.

Example

```
Local Field &MyField;

&MyField = GetField(BUTTON_BOTTOM);
&MyField.LabelImage = Image.BTN_REFRESH;
```

LongTranslateValue

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

Value, ShortTranslateValue

MessageNumber

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

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

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

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

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

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

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 **SeachDefault** to False disables default processing for the field executing the property.

SearchDefault is effective only when used in SearchInit PeopleCode programs.

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

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 SeachDefault 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.

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

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

Value

LongTranslateValue

ShowRequiredFieldCue

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

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

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

DisplayFormat

Style

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 ***override*** 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

This property returns the type of field. The values can be one of the following strings:

- CHAR
- CONTENTREFERENCE
- DATE
- DATETIME
- IMAGE
- LONGCHAR
- NUMBER
- SIGNEDNUMBER
- TIME
- IMAGE

This property is read-only.

Example

```
If &CHARACTER.Type = "NUMBER" Then
     /* perform processing */
Else
     /* error processing */
End-If;
```

Value

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.

Considerations When Checking for Values

If you have an edit-box field, and if the end-user selects the value in it and deletes the value, leaving the field empty, the value of the field in PeopleCode is *not* an empty (zero-length) string. Instead, it is a string with one space character in it. The following is how to check for this:

```
If (fieldname.Value = " ") Then
```

There is no way to distinguish between cases where the user entered nothing or they entered one space character.

Example

```
&CHARACTER.Value = "Hello";
```

See Also

LongTranslateValue, ShortTranslateValue

Visible

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.

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:

- File object declaration
- Scope of a file object
- File layout
- 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

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

"PeopleCode Built-in Functions," GetFile

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Understanding Objects and Classes in PeopleCode," Assigning Objects

File Layout

PeopleTools supports reading and writing to plain text files, and to files that have a format based on a **File Layout** 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

Plain Text Files

File Layout Examples

PeopleTools PeopleBook: Peoplesoft Integration Tools and Utilities, "File Layouts and Data Interchange"

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 will be 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

Open, SetPosition, GetPosition

"PeopleCode Built-in Functions," GetFile

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

File Layout Examples

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

PeopleTools PeopleBook: 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

IsError, ReadRowset, IgnoreInvalidId

"Record Class," IsEditError, ExecuteEdits

"Message Class," ExecuteEdits

"Field Class," MessageNumber, MessageSetNumber,

File Class Built-in Functions

FileExists

FindFiles

GetFile

File Class Methods

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, and releases all resources connected with the file object and goes 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

Open, IsOpen

"PeopleCode Built-in Functions," GetFile

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 Also

SetFileLayout

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

ReadRowset, SetFileLayout, WriteRowset

"Rowset Class"

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

Open, SetPosition

File Access Interruption Recovery

"PeopleCode Built-in Functions," GetFile

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. 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

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.

Warning! 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 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

IsOpen, SetPosition, GetPosition, Close

File Access Interruption Recovery

"PeopleCode Built-in Functions," GetFile

ReadLine

Syntax

ReadLine(*string*)

Description

The **ReadLine** method reads one line of text from the external file. The line includes the newline character used on the platform where the file is stored, 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

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

ReadRowset, WriteString, WriteLine

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 Also

SetFileLayout

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

CurrentRecord, IgnoreInvalidId, ReadLine, SetFileLayout, WriteRecord, WriteRowset

Multiple File Layouts

ReadRowset Example

"Rowset Class," IsEditError

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 CurrentRecord, IsNewFileId, ReadRowset.

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

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

CurrentRecord, IsNewFileId, ReadRowset, SetFileLayout

File Layout Examples

PeopleTools PeopleBook: 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

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

CurrentRecord, CreateRowset, ReadRowset, IgnoreInvalidId, WriteRowset

File Layout Examples

PeopleTools PeopleBook: 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

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

GetPosition, Open

File Access Interruption Recovery

"PeopleCode Built-in Functions", GetFile

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

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

ReadLine, WriteString, WriteRaw

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

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

WriteLine, WriteString

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.

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.

See Also

SetFileLayout

Parameters

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

CreateRowset, ReadRowset, SetFileLayout, WriteRowset

"Record Class"

PeopleTools PeopleBook: 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.

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.

See Also

SetFileLayout

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

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

CreateRowset, ReadRowset, SetFileLayout, WriteRecord

"Rowset Class"

PeopleTools PeopleBook: 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

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

ReadLine, WriteLine, WriteRaw

File Class Properties

In this section, we discuss each File Class property.

CurrentRecord

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

SetFileId, ReadRowset, IsNewFileId

Multiple File Layouts

IgnoreInvalidId

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

ReadRowset

IsError

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

Multiple File Layouts

IsNewFileId

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

ReadRowset, SetFileId, CurrentRecord

IsOpen

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

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;
```

RecTerminator

This property is a file layout property. It returns the string of characters that indicate the end of a file record. Read operations use the value of **RecTerminator** to determine where each file record ends and the next one starts, and write operations append the value of **RecTerminator** to each record written.

This property defaults to the newline character appropriate to the platform where the file is being stored: a linefeed on UNIX systems, and a carriage return/linefeed combination on Windows systems. You need to specify a different **RecTerminator** only if you anticipate that part of the data in a file field will include the newline character used on the storage platform; you must assign to **RecTerminator** a unique string that you know will *not* appear in a file field.

This property is read-write.

Example

```
&MYFILE.RecTerminator = "##EOR##";
```

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 Also

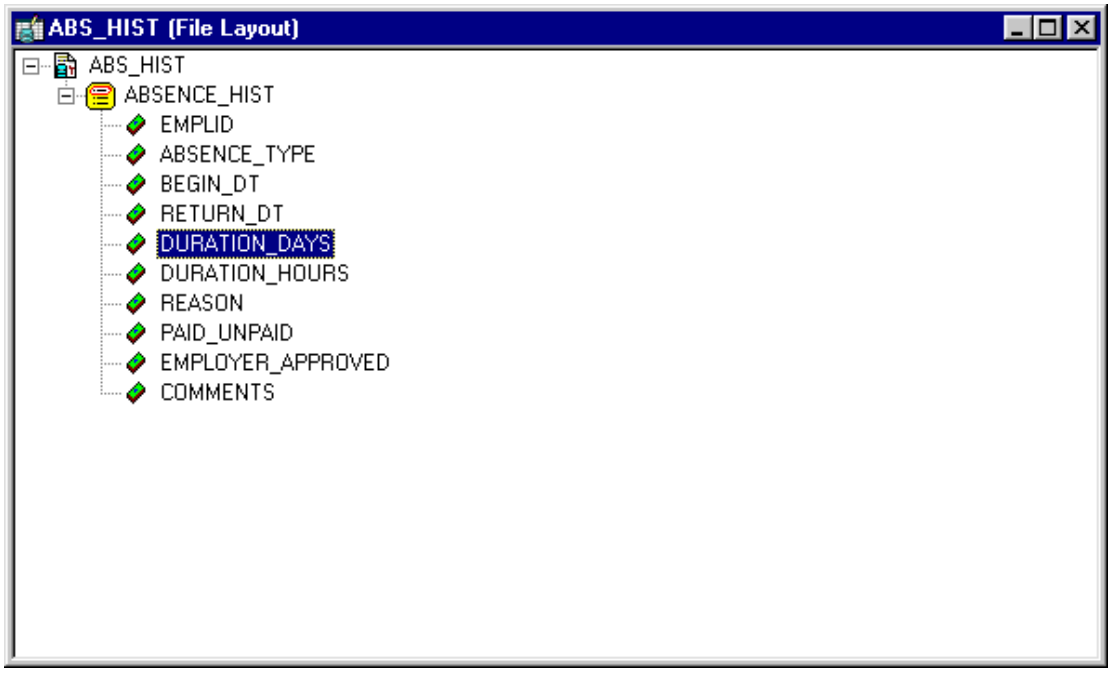
PeopleTools PeopleBook: Peoplesoft Integration Tools and Utilities, "File Layouts and Data Interchange"

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Data Buffer Access"

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using PeopleCode 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;

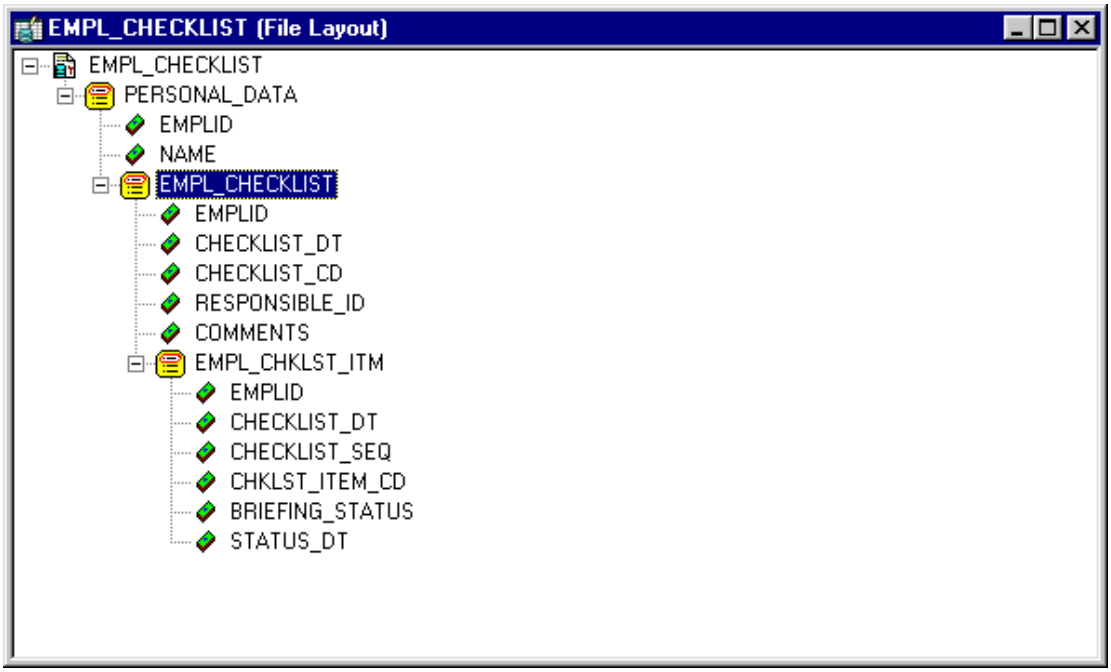
&MYFILE = GetFile("c:\temp\vendor.txt", "R", %FilePath_Absolute);
&REC = CreateRecord(RECORD.ABS_HIST_TEST);
&ARRAY = CreateArrayRept("", 0);

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();
```

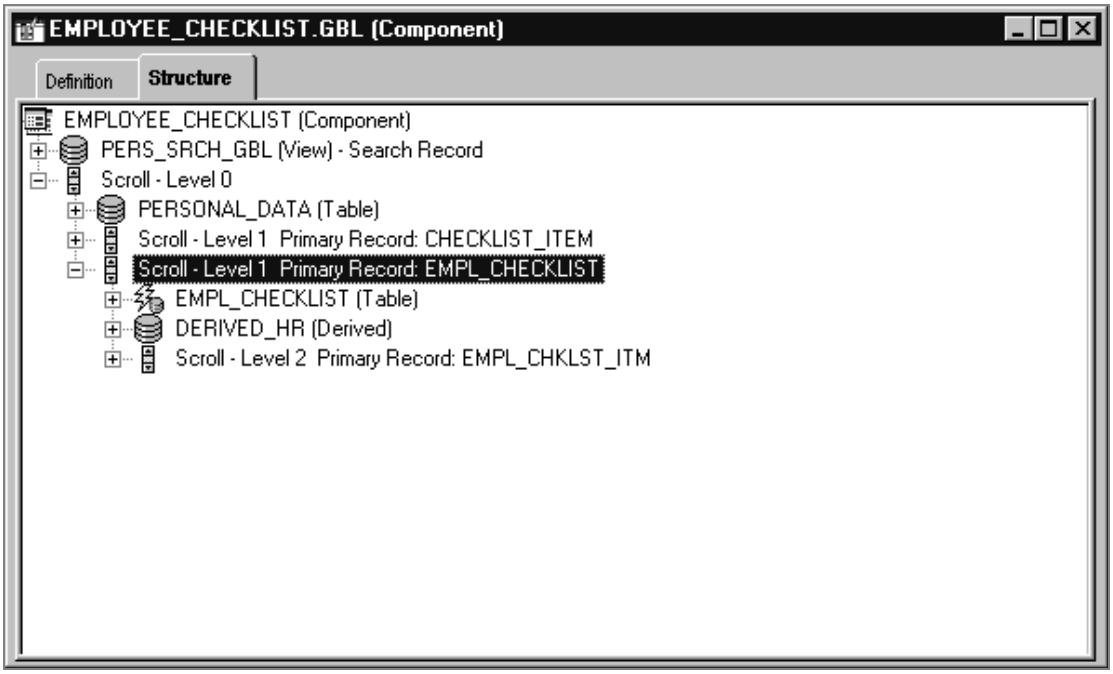
WriteRowset Example

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 Also

"PeopleCode Built-in Functions," GetRowset

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;
```

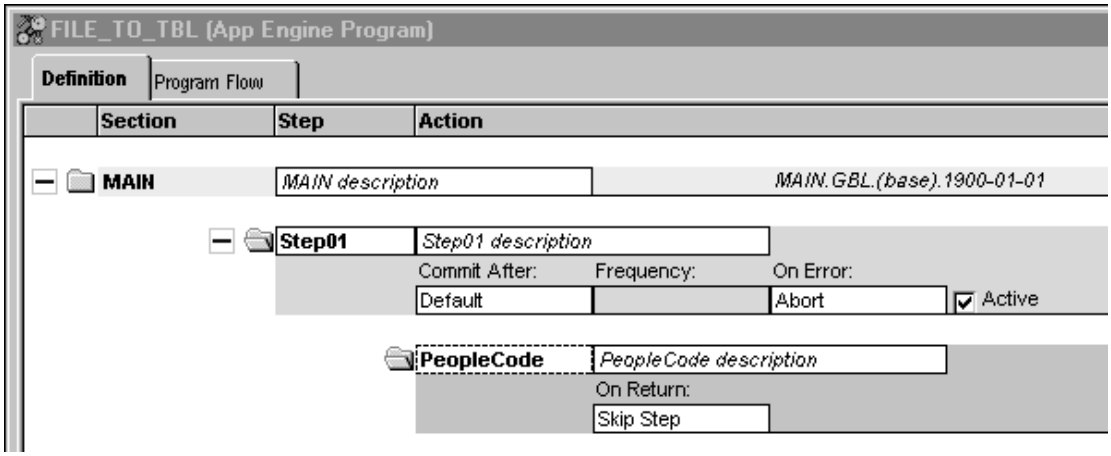
File Rowset Considerations

- 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)");

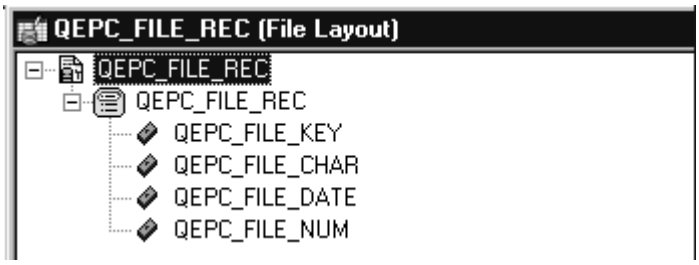
If Not &FILE.IsOpen Then
   Error ("TEST: failed file open");
Else
   If Not &FILE.SetFileLayout(FileLayout.QEPC_FILE_REC) Then
      Error ("TEST: failed SetFilelayout");
   Else
      &FRS = &FILE.ReadRowset();
      While &FRS <> Null
         &FRS.GetRow(1).QEPC_FILE_REC.CopyFieldsTo(&REC);
         &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
```

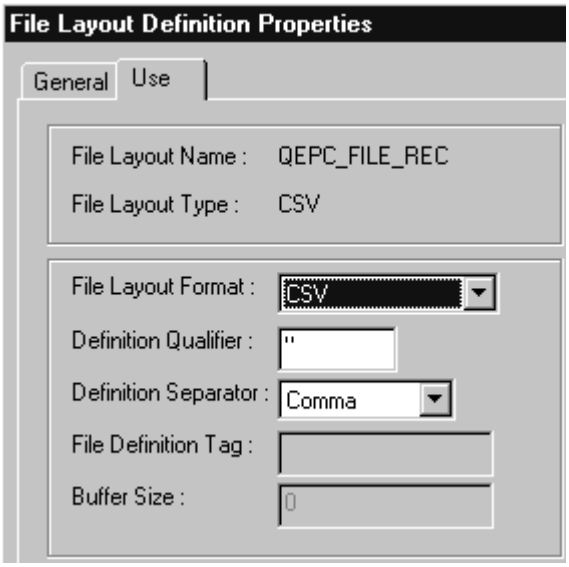
Note that the last field has no qualifier.

The File Layout used to read this record has the following form:



QEPC_FILE_REC File Layout

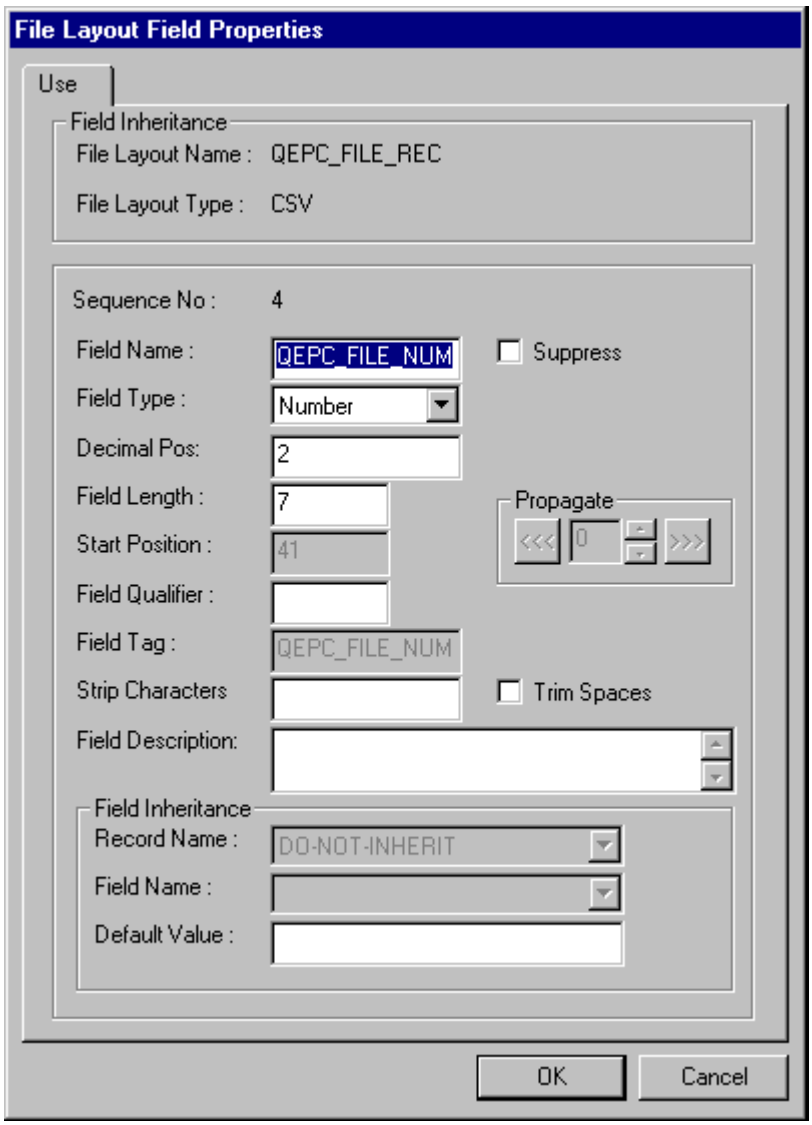
The properties for the QEPC_FILE_REC File Layout are as follows:



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 Also

PeopleTools PeopleBook: 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 Also

SetFileId

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 IsNewFileId is False, you've reached the end of the file.
   - If IsNewFileId 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 &IsNewFileId 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 Also

SetFileId, ReadRowset, IsNewFileId

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 Also

WriteLine, WriteString, WriteRecord

"Record Class"

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

GridColumn Class

PeopleTools PeopleBook: Application Designer, "Chapter Name," Using Grids

PeopleTools PeopleBook: 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

PeopleTools PeopleBook: PeopleCode Developer's Guide, "PeopleCode and the Component Processor," Activate Event

"Rowset Class," HideAllRows

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

GetGrid

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

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

"PeopleCode Built-in Functions", GetGrid

Grid Class Property

In this section, we discuss each Grid class property.

gridcolumn

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);
```

Label

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. **Note.** 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 Also

"PeopleCode Built-in Functions," GetGrid

GridColumn Class Properties

In this section, we discuss each GridColumn class property.

Enabled

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

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. **Note.** 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

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

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.

See Also

PeopleTools PeopleBook: Peoplesoft Application Designer, " Using Page Controls," Using Grids

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;
```

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

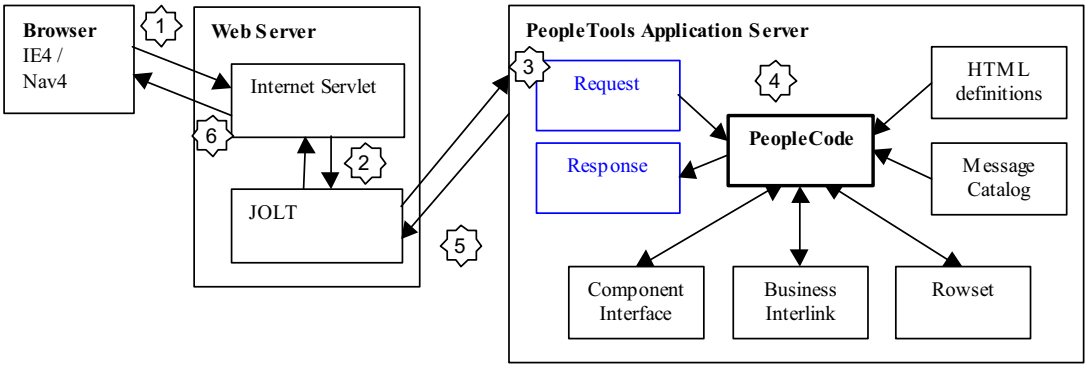
Understanding Internet Script Classes

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.Fie
ldFormula.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.FieldCha
nge.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 Also

PeopleTools PeopleBook: 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.



Hello World

Say this! hi!

You said: hi!

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 Also

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 Also

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

PeopleTools PeopleBook: 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".
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.Fiel
dFormula.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
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	Queyr	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

"PeopleCode Built-in Functions"

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

"Session Class," Error Handling

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 Built-in Functions

The Generate functions are used to generate URLs for iScripts.

See Also

GenerateActGuideContentUrl

GenerateActGuidePortalUrl

GenerateActGuideRelativeUrl

GenerateComponentContentRelURL

GenerateComponentContentURL

GenerateComponentPortalRelURL

GenerateComponentPortalURL

GenerateComponentRelativeURL

GenerateExternalPortalURL

GenerateExternalRelativeURL

GenerateHomepagePortalURL

GenerateHomepageRelativeURL

GenerateQueryContentURL

GenerateQueryPortalURL

GenerateQueryRelativeURL

GenerateScriptContentRelURL

GenerateScriptContentURL

GenerateScriptPortalRelURL

GenerateScriptPortalURL

GenerateScriptRelativeURL

GenerateWorklistPortalURL

GenerateWorklistRelativeURL

Internet Script Reference

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.

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 Also

PeopleSoft Business Interlink Application Developer 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 match 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.

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

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

"System Variables," %AuthenticationToken

ByPassSignOn

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

This property returns a string containing the value the web server passes as the browser platform.

This property is read-only.

BrowserType

This property returns a string describing the browser that sent the request.

This property is read-only.

BrowserVersion

This property returns a string describing the version of the browser that sent the request.

This property is read-only.

ContentURI

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

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

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.FieldFormul
a.IScript_Test
```

HTTPMethod

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

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

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_. . .
```

PortalURI

This property returns a string containing the portion of the current URI before the portal name, and referencing the portal servlet (psp).

This property is read-only.

Example

For the URL:

```
http://localhost/psp/ps84/PORTAL/NODE/e_procurement/fdm/s/WEBLIB. . .
```

This property returns this:

http://localhost/psp/ps84/

Protocol

This property returns the protocol being used for this request as a string of the 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

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

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

This property returns the IP address of the agent that sent the request.

This property is read-only.

RemoteHost

This property returns the fully qualified host name of the agent that sent the request.

This property is read-only.

RequestURI

This property returns the URI, without the protocol.

This property is read-only.

Example

This example uses the following URL:

```
http://serverx/servlets/psp/ps84/PORTAL/NODE/fdm/s/WEBLIB_Portal. . .
```

From the following code

```
&MyRequestURI = &Request.RequestURI;
```

&MyRequestURI contains the following:

```
/servlets/psp/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

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

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

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

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.

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

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.HelloWorld_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

PeopleTools PeopleBook: 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

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

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 PeopleBook: 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

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

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	*Meaning*
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

This property returns the name of the cookie as a string.

This property is read-only.

Path

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

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

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

The Sun website http://java.sun.com/index.html.

Supported Versions of Java

PeopleSoft supports Java version 1.3 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:

- 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

Considerations When Using the PeopleCode Java Functions

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

"System Variables"

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);
```

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

PeopleCode Type	*Java Type/Class*
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
DateTime	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

PeopleTools PeopleBook: 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

The following are the PeopleCode Java built-in functions:

CreateJavaArray

CreateJavaObject

GetJavaClass

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

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Messages"

PeopleTools PeopleBook: 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

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining 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

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Messages"

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 Include check 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

"Record Class," SetDefault

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

ExecuteEdits

"Record Class," ExecuteEdits

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Sending and Receiving Messages," Processing Inbound Errors

Message Class Built-In Functions

AddSystemPauseTimes

CreateMessage

DeleteSystemPauseTimes

GetMessage

GetSyncLogData

PingNode

ReturnToServer

SetChannelStatus

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

CopyRowset

"PeopleCode Built-in Functions," CreateMessage

PeopleTools PeopleBook: 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 CopyRowsetDelta for more details.

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

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

CopyRowsetDelta

"PeopleCode Built-in Functions," GetRowset

"Rowset Class," Fill

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

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

PeopleTools PeopleBook: Integration Tools, "Chapter Name," Understanding PSCAMA.

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Understanding Objects and Classes in PeopleCode," Assigning Objects.

CopyRowset

"Rowset Class"

"PeopleCode Built-in Functions," GetRowset

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.

User Action	*CopyRowsetDeltaOriginal Output*	*Comments*
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.
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

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

PeopleTools PeopleBook: Integration Tools, "Chapter Name," Understanding PSCAMA.

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Understanding Objects and Classes in PeopleCode," Assigning Objects.

CopyRowset, CopyRowsetDelta

"Rowset Class"

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 Also

"Field Class," EditError, MessageNumber, MessageSetNumber

"Record Class," ExecuteEdits

SetEditTable

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

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

SetEditTable

"Field Class," EditError, MessageNumber, MessageSetNumber

"Record Class," IsEditError

PeopleTools PeopleBook: 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

LoadXMLString

GetQueryString

Syntax

```
GetQueryString(Parameter_Name)
```

Description

Use the **GetQueryString** method to obtain the parameter values of a query string.

Parameters

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

SetQueryString

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

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

Clone, **CreateMessage**

"Rowset Class"

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Understanding PeopleCode Objects and Classes," Assigning Objects

LoadXMLString

Syntax

```
LoadXMLString(XMLstring)
```

Description

The **LoadXMLString** function loads the message object executing the method with the XML string *XMLstring*.

Parameters

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

GenXMLString

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

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

IsActive

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Messages"

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

%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

ExecuteEdits

"Field Class," EditError, MessageNumber, MessageSetNumber

"Record Class," IsEditError, SetEditTable

PeopleTools PeopleBook: 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_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

GetQueryString

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

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

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Messages"

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

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

Publish

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. **Note.** The **Update** method can't be called from subscription PeopleCode.

Parameters

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

GetRowset

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Messages"

Message Class Properties

In this section, we discuss the Message class properties. The properties are discussed in alphabetical order.

ChannelName

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

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Messages"

Cookies

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.

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;
```

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

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

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

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

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

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Messages"

"PeopleCode Built-in Functions", IsMessageActive

IsDelta

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

CopyRowsetDelta

IsEditError

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

ExecuteEdits

"Field Class," MessageNumber, MessageSetNumber

"PeopleCode Built-in Functions," Exit

PeopleTools PeopleBook: Integration, "Sending and Receiving Messages," Processing Inbound Errors

IsEmpty

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

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

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

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

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

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Defining Messages," Understanding Nonrepudiation

PubID

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

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

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

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

"System Variables," %MaxMessageSize

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Sending and Receiving Messages"

SubName

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

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

- Using the "Run PeopleCode" option from Message Subscription is *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

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Sending and Receiving Messages"

CHAPTER 18

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

PeopleTools PeopleBook: Peoplesoft Workflow, "Introduction to 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

"Application Classes"

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

Notification Classes Constructors

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

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

Notification Class

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

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

NotificationAddress Class

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

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

NotificationTemplate Class

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

WorklistEntry Class

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 Also

Notification Classes Examples

PeopleTools PeopleBook: Peoplesoft Workflow, "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

PeopleTools PeopleBook: Peoplesoft Workflow, "Introduction to Workflow"

Notification Class Properties

In this section, we discuss the Notification class properties. The properties are discussed in alphabetical order.

ContentType

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

Message

dttmCreated

This property returns the Datetime the notification created as a datetime value. Together with NotifyFrom this makes a notification unique.

See Also

NotifyFrom

EmailReplyTo

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

Send, NotifyFrom

FileNames

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

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

FileNames

language_cd

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

This property returns the text of the message sent with the notification as a string.

This property is read-write.

NotifyBCC

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

NotifyCC

NotifyCC

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

NotifyBCC

NotifyFrom

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

This property returns the GUID of the notification. This is a unique identifier used for tracking purposes only.

This property is read-write.

NotifyTo

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

NotifyCC, NotifyBCC

SourceComponent

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

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

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

This property returns the subject used in the email notification as a string.

This property is read-write.

Template

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 Also

Notification Classes Examples

PeopleTools PeopleBook: Peoplesoft Workflow, "Notification Templates"

NotificationAddress Class Properties

In this section, we discuss the NotificationAddress class properties. The properties are discussed in alphabetical order.

Channel

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

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

This property returns the Email Address of the entity to be notified.

This property is read-write.

Language

This property returns the language code of the entity to be notified.

This property is read-write.

Oprid

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 Also

PeopleTools PeopleBook: Peoplesoft Workflow, "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

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

SetupCompVarsAndRcpts, SetupGenericVars

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

&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

GetAndExpandTemplate

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

&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

GetAndExpandTemplate

NotificationTemplate Class Properties

In this section, we discuss the NotificationTemplate class properties. The properties are discussed in alphabetical order.

ComponentId

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

This property returns the notification instructions as a string.

This property is read-write.

Language

This property returns the language used to retrieve template descriptions, as a string.

This property is read-write.

Market

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

This property returns the priority of the notification template as a string.

This property is read-write.

Responses

This property is populated only if there are any RIMM responses.

This property is read-write.

Subject

This property returns the subject of the notification template as a string.

This property is read-write.

TemplateId

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

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

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 Also

Creating a WorklistEntry

Updating a WorklistEntry

PeopleTools PeopleBook: Peoplesoft Workflow, "Introduction to 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

busactivity, buseventname, busprocname, instanceid, transactionid

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

Update

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

Save

WorklistEntry Properties

In this section, we discuss the WorklistEntry class properties. The properties are discussed in alphabetical order.

actiondttm

This property returns the Worklist action datetime as a datetime.

This property is read-write.

busactivity

This property returns the source activity name of the source business process as a string.

This property is read-write.

buseventname

This property returns the source event name of the source activity as a string.

This property is read-write.

busprocname

This property returns the source business process name as a string.

This property is read-write.

commentshort

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

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

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

This property returns the datetime that the worklist was selected to be worked as a datetime value.

This property is read-write.

instatus

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

This property returns the datetime the worklist entry timed out. Only some worklist types use this property.

This property is read-write.

instworkeddttm

This property returns the datetime worklist was worked as a datetime value.

This property is read-write.

oprid

This property returns the UserID that is assigned the worklist as a string.

This property is read-write.

originatorid

This property returns the UserID that caused the worklist to be created as a string.

This property is read-write.

prevoprid

This property returns the previous UserID who used this worklist entry as a string.

This property is read-write.

syncid

This property is reserved for future use.

This property is read-write.

timedout

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

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

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

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

This property returns the number of days to select worklist as a number.

This property is read-write.

wldaystowork

This property returns the number of days to work the worklist as a number.

This property is read-write.

worklistname

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 Also

PeopleTools PeopleBook: Peoplesoft Workflow, "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
   - 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 19

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 shouldn't be used until after the page processor has loaded the page: don't instantiate this object in RowInit PeopleCode, use it in PostBuild or Activate instead.

See Also

PeopleTools PeopleBook: 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 shouldn't be used until after the page processor has loaded the page: don't instantiate this object in RowInit PeopleCode. Use it in PostBuild or Activate instead.

Page Class Built-in Function

GetPage

Page Class Properties

In this section, we discuss each Page class properties.

DisplayOnly

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 can't 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

This property returns a reference to the page name definition (a string value.)

This property is read-only.

Example

```
&MYPAGENAME = &MYPAGE.Name;
```

Visible

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.

Note. You can't hide the current page.

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.

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

PeopleTools PeopleBook: Peoplesoft Application Designer, "Creating Component Definitions"

CHAPTER 20

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 PeopleBook: Peoplesoft Portal Technology, "Introducing PeopleSoft Portal Technology", "Understanding the Registry and URLs", "Administering a PeopleSoft Portal"

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

Adding a Folder

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

PeopleTools PeopleBook: Peoplesoft Integration Broker, "Administering Basic Integrations," Configuring a Node

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 Using Attributes for an example of using attributes.

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.

See Also

Cascade

PeopleTools PeopleBook: Security, "Working With Permission Lists".

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.

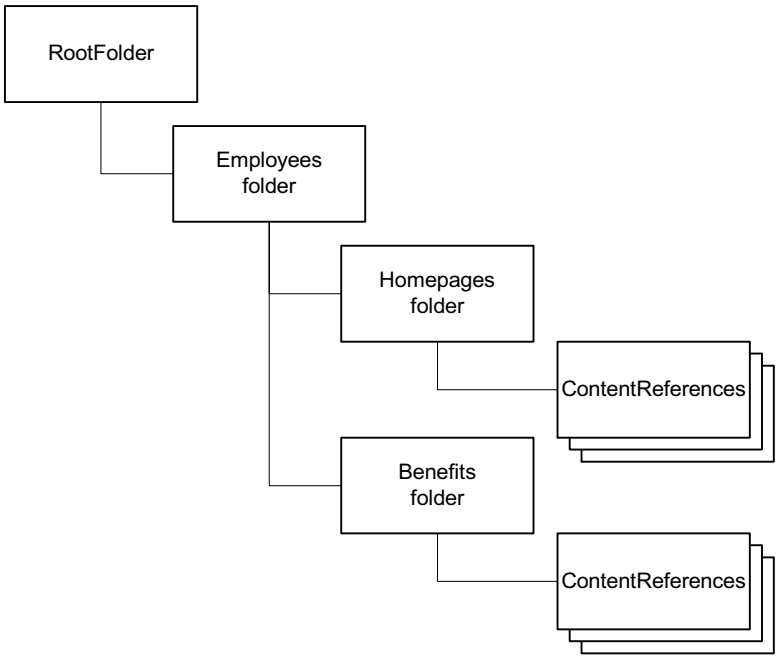
Note. You *cannot* add any PermissionValues to a collection returned by the CascadedPermissions property. You can add values only to the collection returned by the Permissions property.

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

Setting Permissions Using the PermissionValue Object

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

"Session Class," Error Handling

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.

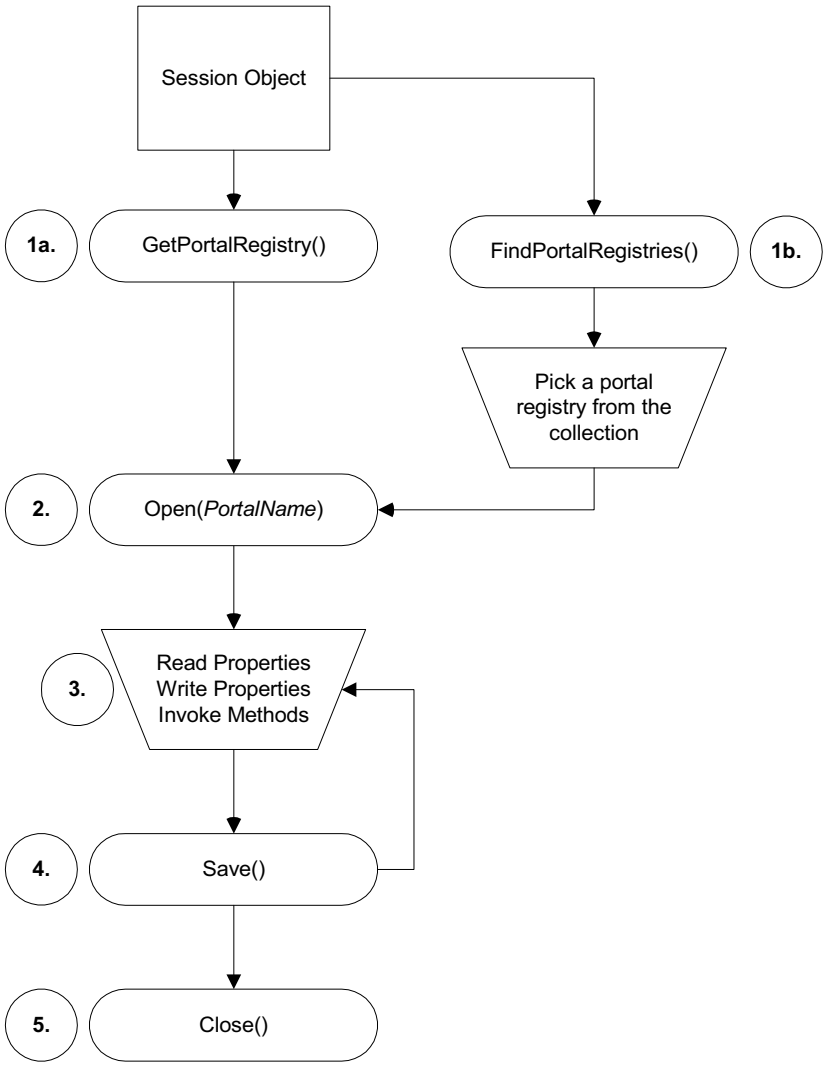
4. After you make your changes, you must save your work. The Save method must be executed at the appropriate level, such as at the folder level for changes to a folder, at the PortalRegistry level for changes to a node, and so on.

    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 Saving Content Considerations.

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

PortalRegistry Classes Example

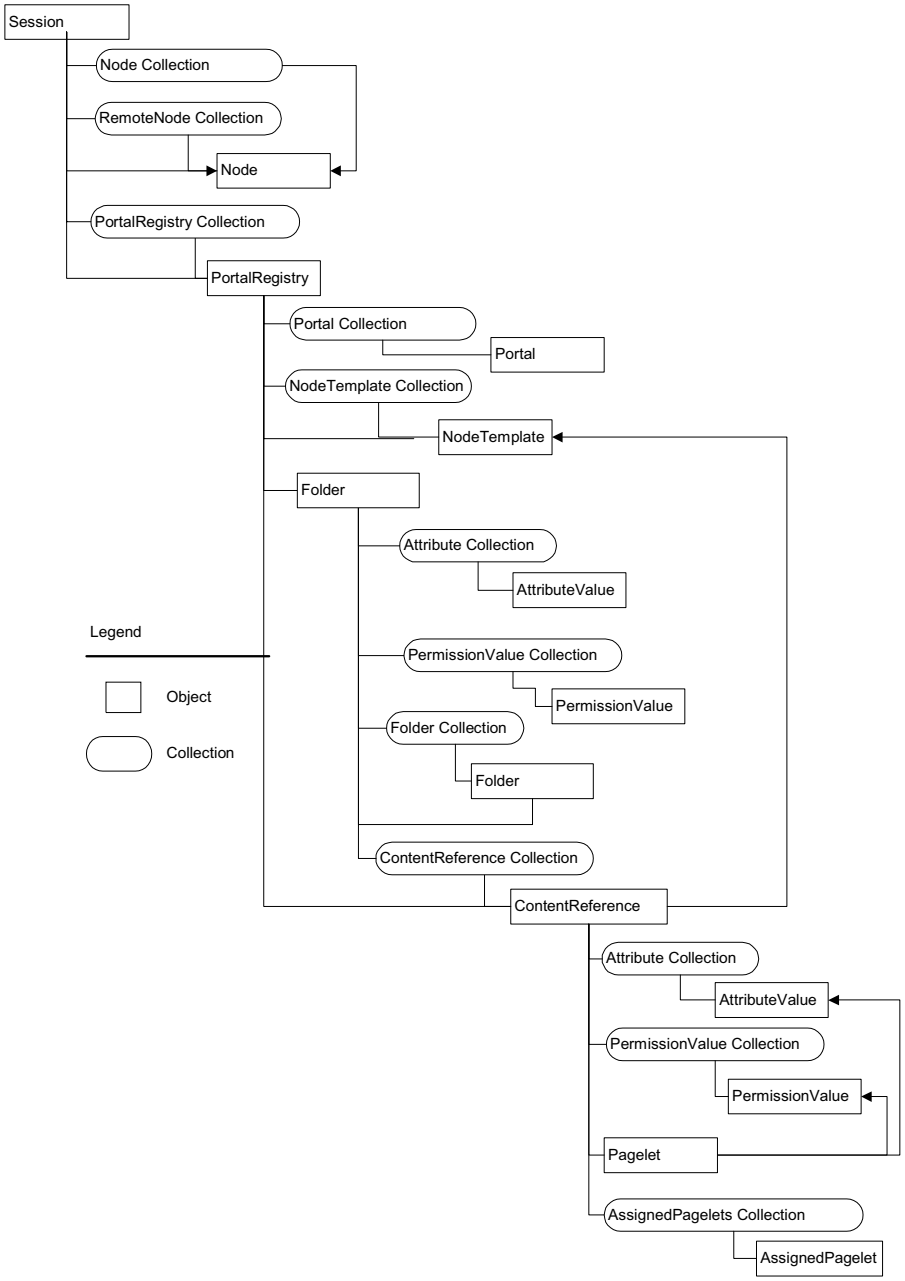
Viewing the PortalRegistry Class Hierarchy

There are many different classes used with the PortalRegistry API. The following flowchart shows the different classes, and how they are interrelated.

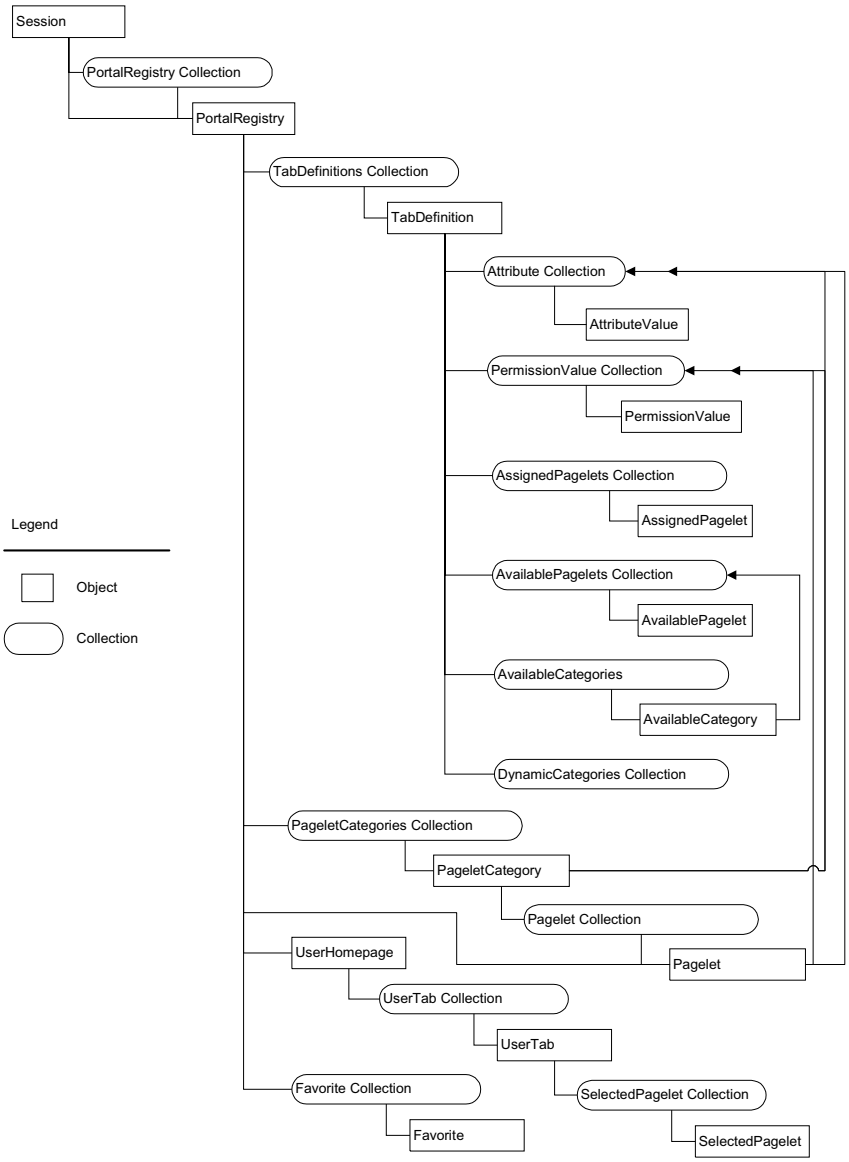
Hierarchy Considerations

- The 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

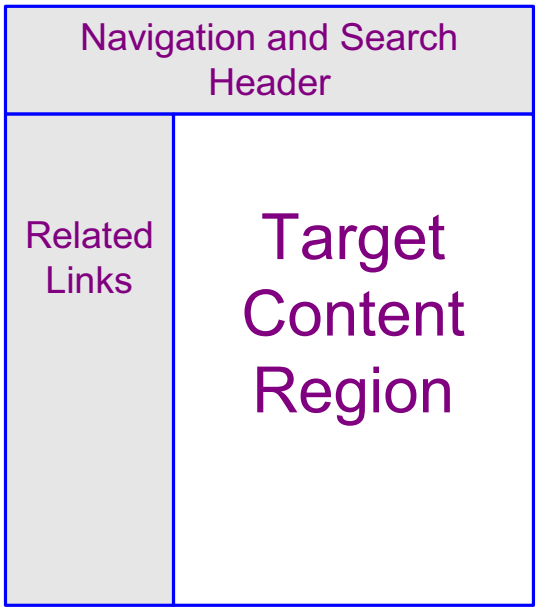
Target

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 Also

PeopleTools PeopleBook: Peoplesoft Portal 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

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 PeopleBook: Peoplesoft Portal Technology, "Understanding the Registry and URLs"

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.

See Also

PeopleTools PeopleBook: Peoplesoft Portal Technology, "Understanding the Registry and URLs"

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
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)

UsageType	*TemplateType*	*StorageType*	*URLType*
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. ***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 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 PortalRegistry Classes Example for example programs of how to access the PortalRegistry objects using language environments other than PeopleCode.

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 PeopleBook: Peoplesoft Portal Technology, "Introducing 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

"Session Class"

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

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

Node Class

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

Node Collection

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

RemoteNode Collection

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

GetPortalRegistry, PortalRegistry Collection Methods

Changing PortalRegistry Properties

"Search Classes"

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

Open, Save

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

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

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

Open, Save, Close, Delete

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

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

Create, Portals

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

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

Close, Open

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

UserHomepage Class

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

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

Content Reference Class

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.

See Also

PeopleTools PeopleBook: Peoplesoft Portal Technology, "Understanding the Registry and URLs"

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

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

Content Reference Class, GetQualifiedURL, QualifiedURL

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

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

Folder Class

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

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

Pagelet Class

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

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.

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

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

RevokePermissionForComponent

PeopleTools PeopleBook: 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

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

RevokePermissionForComponent

PeopleTools PeopleBook: 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

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

Open, Save, Close, Delete

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

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

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

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

GrantPermissionForComponent

PeopleTools PeopleBook: 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

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

GrantPermissionForScript

PeopleTools PeopleBook: 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

Save folder method, Save content reference method.

PortalRegistry Class Properties

In this section, we discuss the PortalRegistry class properties. The properties are discussed in alphabetical order.

DefaultTemplate

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

TemplateObject

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

This property returns a reference to the Favorites Collection for the current end-user.

This property is read-only.

See Also

Favorite Collection

Homepage

This property returns a reference to the Homepage for the current end-user.

This property is read-only.

Name

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

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

NodeTemplate Collection

OwnerId

This property returns or sets the owner ID of the PortalRegistry object as a string.

This property is read-write.

PageletCategories

This property returns a reference to a PageletCategories Collection that contains all the PageletCategories for a portal.

This property is read-only.

See Also

PageletCategory Collection

Portals

This property returns a reference to all the Portals associated with the PortalRegistry.

This property is read-only.

See Also

Portal Collection

RootFolder

This property returns the root folder object for this PortalRegistry object.

This property is read-only.

TabDefinitions

This property returns reference to a TabDefinitions Collection that contains all the TabDefinitions for a portal.

This property is read-only.

See Also

TabDefinition Collection

TemplateObject

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

DefaultTemplate

PortalRegistry Collection

A PortalRegistry Collection is returned by the FindPortalRegistries session class method.

See Also

FindPortalRegistries

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

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

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 Also

GetLocalNode, Node Collection, RemoteNode Collection

Node Class Properties

In this section, we discuss the Node class properties. The properties are discussed in alphabetical order.

ActiveNode

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

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

NodeType

ContentURI

This property returns the URI for the content webserver of this node as a string.

This property is read-only.

DefaultPortalName

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

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

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

NodeType

IsLocal

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

NodeType

Name

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

This property returns the password for this node object as a string.

This property is read-only.

NodeType

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

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

NodeType

ToolsRelease

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

NodeType

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 Also

GetNodes

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

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<i>Node</i> = &MyCollection.Next();
```

Node Collection Property

In this section, we discuss the Count property.

Count

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 Also

GetRemoteNodes

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

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

```
&MyNode = &MyCollection.Next();
```

RemoteNode Collection Property

In this section, we discuss the Count property.

Count

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 Also

Portal Collection

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

Save PortalRegistry method

Portal Class Properties

In this section, we discuss the Portal class properties. The properties are discussed in alphabetical order.

HostNodeName

This property sets or returns the name of the node hosting the portal as a string.

This property is read-write.

IsLocal

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

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 Also

Portals

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

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

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 Also

NodeTemplate Collection

NodeTemplate Class Properties

In this section, we discuss the NodeTemplate class properties. The properties are discussed in alphabetical order.

DefaultTemplate

This property specifies the name of the template to be applied to the node.

This property is read-write.

Name

This property specifies the name of node that the template is to be applied to.

This property is read-write.

TemplateObject

This property returns a reference to the template object associated with this node as a content reference.

This property is read-only.

See Also

Content Reference Class

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 Also

NodeTemplates

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

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

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

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

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 Also

RootFolder, FindFolderByName, Folder Collection

Adding a Folder

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

Save PortalRegistry method, Save content reference method.

Folder Class Properties

In this section, we discuss the Folder class properties. The properties are discussed in alphabetical order.

Attributes

This property returns an Attribute Collection containing the AttributeValue objects for this folder.

This property is read-only.

See Also

Attribute Collection

Author

This property returns the author (PeopleSoft user ID) for this folder as a string.

This property is read-only.

AuthorAccess

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

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

FindFolderByName

CascadedPermissions

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

PermissionValue Collection, Permissions

ContentRefs

This property returns the ContentReference Collection for this folder.

This property is read-only.

See Also

Content Reference Collection

CreationDate

This property returns the creation date for this folder as a string.

This property is read-only.

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

This property returns a reference to the Folder Collection for this folder.

This property is read-only.

See Also

Folder Collection

IsMobile

This property returns True if the folder is used with mobile applications, False otherwise.

This property is read-write.

IsVisible

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

ValidFrom, ValidTo

Label

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

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

Label

OwnerId

This property returns the name of the owner for this folder as a string.

This property is read-write.

ParentName

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

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

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

PermissionValue Collection, CascadedPermissions

Product

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

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

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

This property returns or sets the date this folder is valid from as a string.

This property is read-write.

ValidTo

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 Also

Folders

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

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

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

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

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 Also

FindCRefByURL, FindCRefByName, Content Reference Collection

Adding a Content Reference

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

Save PortalRegistry method, Save Folder method.

Content Reference Class Properties

In this section, we discuss the Content Reference class properties. The properties are discussed in alphabetical order.

AbsoluteContentURL

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?pag
e=CUST_DATA1&&Action=U&emplid=00001
```

AbsolutePortalURL

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?pag
e=CUST_DATA1&&Action=U&emplid=00001
```

AssignedPagelets

This property returns an AssignedPagelet collection.

This property is read-only.

See Also

AssignedPagelet Collection

Attributes

This property returns an Attribute Collection containing the AttributeValues for this content reference object.

This property is read-only.

See Also

Attribute Collection

Author

This property returns the author (PeopleSoft user ID) for this content reference object as a string.

This property is read-only.

AuthorAccess

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

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

FindCRefByURL, FindCRefByName

CascadedPermissions

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

PermissionValue Collection, Permissions

ContentProvider

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

This property returns the creation date for this content reference object as a string.

This property is read-only.

Data

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

StorageType

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

This property returns the HTML text associated with this content reference as a string.

This property is read-only.

IsMobile

This property returns True if this content reference is used with mobile applications, False otherwise.

This property is read-write.

IsVisible

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

ValidFrom, ValidTo

Label

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

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

This property returns or sets the owner ID of the content reference object as a string.

This property is read-write.

ParentName

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

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

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

PermissionValue Collection, CascadedPermissions

Product

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

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

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.

RelativeURL

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

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

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.

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

Data

Template

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

UsageType, TemplateType, TemplateObject

TemplateObject

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

Template

TemplateType

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

Using Content References

URL

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

AbsoluteContentURL

Using Content References

PeopleTools PeopleBook: Peoplesoft Portal Technology, "Understanding the Registry and URLs"

URLType

This property indicates what kind of PeopleCode definition is used to build the page for the content reference. This property is used only for content that has the StorageType property set as **Remote**. If the type is External, a PeopleCode definition is *not* used. This property takes a string value.

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

UPGE is the default value on a new content reference.

This property is read-write.

See Also

StorageType

UsageType

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

Using Content References

ValidFrom

This property returns or sets the date this content reference is valid from as a string.

This property is read-write.

ValidTo

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 Also

ContentRefs

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

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

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

Using Content References

ItemByName

Syntax

ItemByName(*Name*)

Description

The **ItemByName** method returns the content reference object with the name *Name*.

Parameters

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

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 Also

Attribute Collection Methods

Using Attributes

AttributeValue Class Properties

In this section, we discuss the AttributeValue class properties. The properties are discussed in alphabetical order.

Label

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.

Name

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

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

Label

Value

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 Also

Attributes Content Reference property

Attributes Folder property

Attributes Pagelet Category property

Attributes Pagelet property

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

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

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

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

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 Also

PermissionValue Collection

Using Security

Setting Permissions Using the PermissionValue Object

PermissionValue Class Properties

In this section we discuss the Cascade and Name properties.

Cascade

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

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 Also

CascadedPermissions, Permissions folder properties

CascadedPermissions, Permissions content reference properties

CascadedPermissions, Permissions PageletCategory object properties

CascadedPermissions, Permissions Pagelet properties

Using Security

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

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

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

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

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 Also

TabDefinition Collection

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

Save PortalRegistry method, Save Folder method, Save ContentReference method

TabDefinition Class Properties

In this section, we discuss the TabDefinition class properties. The properties are discussed in alphabetical order.

AssignedPagelets

This property returns a reference to an AssignedPagelet Collection for this TabDefinition object.

This property is read-only.

See Also

AssignedPagelet Collection

AvailableCategories

This property returns a reference to an AvailableCategory Collection for this TabDefinition object.

This property is read-only.

See Also

AvailableCategory Collection

AvailablePagelets

This property returns a reference to an AvailablePagelet Collection for this TabDefinition object.

This property is read-only.

See Also

AvailablePagelet Collection

Attributes

This property returns an Attribute Collection containing the AttributeValues for this TabDefinition object.

This property is read-only.

See Also

Attribute Collection

Author

This property returns the author (PeopleSoft user ID) for this TabDefinition object as a string.

This property is read-only.

AuthorAccess

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

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:

String Value	*Description*
2	Two-column table
3	Three-column table

This property is read-write.

CreationDate

This property returns the creation date for this TabDefinition object as a string.

This property is read-only.

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

This property returns a reference to a DynamicCategory Collection for this TabDefinition object.

This property is read-only.

See Also

DynamicCategory Collection

HelpID

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

This property returns the HTML text associated with this tab definition as a string.

This property is read-only.

IsLayoutLocked

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

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

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

This property indicates whether the user can change when and how the tab displays. This property takes a string value.

The values are:

String 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.

String Value	*Description*
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

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

This property returns or sets the owner ID of the TabDefinition object as a string.

This property is read-write.

Product

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

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

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

URL

PeopleTools PeopleBook: Peoplesoft Portal Technology, "Understanding the Registry and URLs"

SequenceNumber

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

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

This property returns or sets the date this TabDefinition is valid from as a string.

This property is read-write.

ValidTo

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 Also

TabDefinitions

TabDefinition Collection Methods

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

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

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

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

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 Also

AssignedPagelet Collection

AssignedPagelet Class Properties

In this section, we discuss the AssignedPagelet class properties. The properties are discussed in alphabetical order.

Column

This property returns the column in which this pagelet is displayed, as a number.

This property is read-write.

LayoutBehavior

This property indicates whether the user can change when and how the pagelet displays. This property takes a string value.

The values are:

String 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

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

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 Also

AssignedPagelets

AssignedPagelet Class

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

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

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:

String 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.

String Value	Description
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

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

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 Also

AvailableCategory Collection

AvailableCategory Class Properties

In this section, we discuss the AvailableCategory class properties. The properties are discussed in alphabetical order.

AvailablePagelets

This property returns a reference to an AvailablePagelet Collection associated with the specified CategoryName.

This property is read-only.

See Also

AvailablePagelet Collection

CategoryName

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 Also

AvailableCategory Collection

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

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

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 Also

AvailablePagelet Collection

AvailablePagelet Class Properties

In this section, we discuss the AvailablePagelet class properties. The properties are discussed in alphabetical order.

CategoryLabel

This property turns the label of the category to which the pagelet is assigned, as a string.

This property is read-only.

CategoryName

This property returns the name of the category to which the pagelet is assigned, as a string.

This property is read-only.

Column

This property returns the column in which this pagelet is displayed, as a number.

This property is read-only.

LayoutBehavior

This property indicates whether the user can change when and how the pagelet displays. This property takes a string value.

The values are:

String 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.

String Value	Description
1FIX	The user can't remove this pagelet or change its position on the tab.

This property is read only.

PageletLabel

This property returns the label for this property as a string.

This property is read-only.

PageletName

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

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 Also

AvailablePagelets

AvailablePagelet Class

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

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

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 Also

DynamicCategories

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

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

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

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

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 Also

PageletCategory Collection

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

Save PortalRegistry method, Save Folder method.

PageletCategory Class Properties

In this section, we discuss the PageletCategory class properties. The properties are discussed in alphabetical order.

Attributes

This property returns an Attribute Collection containing the AttributeValues for this PageletCategory object.

This property is read-only.

See Also

Attribute Collection

Author

This property returns the author (PeopleSoft user ID) for this PageletCategory as a string.

This property is read-only.

AuthorAccess

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

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

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). **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

PermissionValue Collection

Permissions

CreationDate

This property returns the creation date for this PageletCategory as a string.

This property is read-only.

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

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

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

Label

OwnerId

This property returns or sets the owner ID of the PageletCategory object as a string.

This property is read-write.

Pagelets

This property returns a reference to a Pagelet Collection for this PageletCategory.

This property is read-only.

See Also

Pagelet Collection

Permissions

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

PermissionValue Collection

Product

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

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

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 Also

PageletCategories

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

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

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

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

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 Also

Pagelet Collection

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

Save PortalRegistry method, Save ContentReference method, Save PageletCategory method

Pagelet Class Properties

In this section, we discuss the Pagelet class properties. The properties are discussed in alphabetical order.

Attributes

This property returns an Attribute Collection containing the AttributeValues for this pagelet object.

This property is read-only.

See Also

Attribute Collection

Author

This property returns the author (PeopleSoft user ID) for this pagelet object as a string.

This property is read-only.

AuthorAccess

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

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

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

PermissionValue Collection

Permissions

ContentProvider

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

This property returns the creation date for this pagelet object as a string.

This property is read-only.

DefaultColumn

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

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

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

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

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.

Label

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

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

This property returns or sets the owner ID of the pagelet object as a string.

This property is read-write.

ParentName

This property returns the parent folder name for this pagelet object as a string.

This property is read-only.

Permissions

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

PermissionValue Collection, CascadedPermissions

Product

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

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

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

URL

PeopleTools PeopleBook: Peoplesoft Portal Technology, "Understanding the Registry and URLs"

SequenceNumber

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

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.

See Also

PeopleTools PeopleBook: Peoplesoft Portal Technology, "Understanding the Registry and URLs"

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 Also

Pagelets

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

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

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

Using Content References

ItemByName

Syntax

```
ItemByName(Name)
```

Description

The **ItemByName** method returns the pagelet object with the name *Name*.

Parameters

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

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 Also

Homepage

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

Folder Save method, PortalRegistry Save method

UserHomepage Properties

In this section, we discuss the UserHomepage properties. The properties are discussed in alphabetical order.

Greeting

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

This property returns the UserId for this UserHomepage as a string.

This property is read-only.

UserTab

This property returns a reference to a UserTab collection.

This property is read-only.

See Also

UserTab Collection

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 Also

UserTab Collection

UserTab Class Properties

In this section, we discuss the UserTab class properties. The properties are discussed in alphabetical order.

ColumnLayout

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

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

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

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

SelectedPagelet Collection

SequenceNumber

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

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 Also

UserTab

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

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

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

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

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 Also

SelectedPagelet Collection

SelectedPagelet Class Properties

In this section, we discuss the SelectedPagelet class properties. The properties are discussed in alphabetical order.

CategoryName

This property returns the category of the pagelet as a string.

This property is read-only.

Column

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

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

This property returns the name of the pagelet as a string.

This property is read-only.

Row

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 Also

SelectedPagelets

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

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

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

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

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 Also

Favorite Collection

Favorite Class Properties

In this section, we discuss the Favorite class properties. The properties are discussed in alphabetical order.

CRefName

This property returns the name of the content reference as a string.

This property is read-only.

Label

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

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

This property returns the sequence number used to order favorites.

This property is read-write.

URL

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=DEFAUL
T&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 Also

Favorites

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

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

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

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

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 Also

"Session Class," Error Handling

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 Also

"Session Class," Error Handling

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.

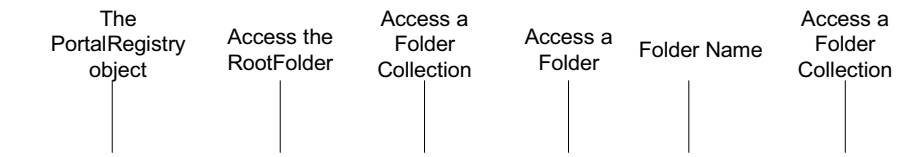
```
If NOT &MyPortal.Open(MYRECORD.PORTAL_NAME) Then
     /* Do error handling */
End-if;
```

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;
```

   To following breaks the previous code down:



PortalRegistry syntax

The second Folders property returns a reference to a folder collection.

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 Also

Using Security

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("Approv
e 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("Approv
e 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 Also

Adding a Folder

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;
```

5. 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");
```

6. 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 Also

Permissions, CascadedPermissions

Using PermissionValue and Cascading Permissions

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

```
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

```
/************************************************************************
*                                                                       *
* 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(hSession);

     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 21

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

PeopleSoft Process Scheduler

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.

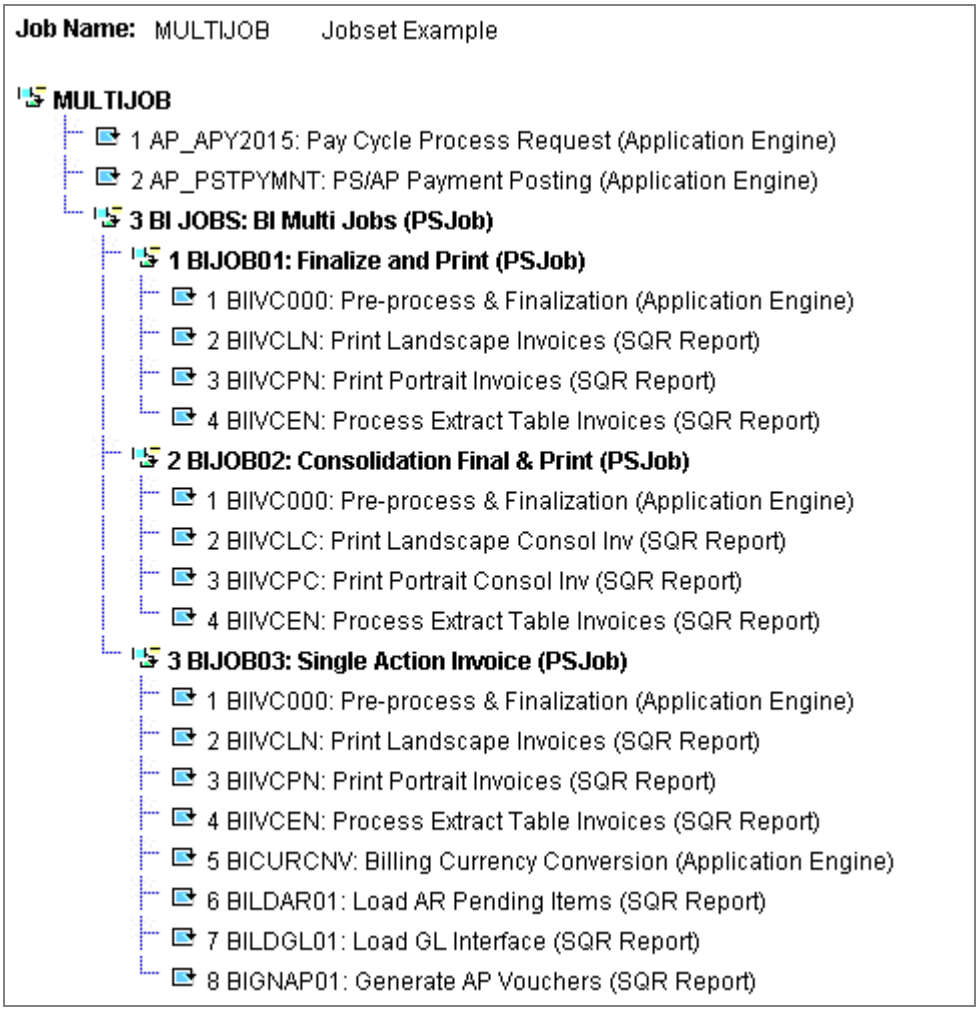
The following is a simple example of a job containing only a single process:



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

Process	*PrcsItemLevel*	*JobName*	*JobSeqNo*
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
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

PeopleTools PeopleBook: Peoplesoft Process Scheduler, "Defining Jobs and JobSets"

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

Generic ProcessType	*OutDestType*	*Valid OutDestFormats*	*Default*
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
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

```
&RQST.OutDestType = "Web";
&RQST.OutDestFormat = "PDF";
&RQST.OutDest = "User : VP1,Role :Managers";
```

Set Attributes For Email

```
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

```
&RQST.SetOutputOption("Web", "PDF", "");
&RQST.AddDistributionOption("User", "QEDMO");
&RQST.AddDistributionOption("Role", "MANAGERS");
```

Set Attribute for Email

```
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 Function

CreateProcessRequest

GetNextProcessInstance

SetupScheduleDefnItem

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

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

SetEmailOption, SetOutputOption

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

PrintJobRqstRpt, PrintSchdlHTMLRpt

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

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: 3MIX - 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\SERVERWINX86\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)

2	4201	AEMINITEST	Simple AE test program	Application Engine	Success	TEST	Web	Text Files (*.txt)	PSNT2	2002 - 01 - 08 -	

Example HTML report

See Also

PrintJobHTMLRpt, PrintSchdlHTMLRpt

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

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

PrintJobHTMLRpt, PrintJobRqstRpt

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

Status

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 following properties:
  - 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

RunControlID, JobName, ProcessName, ProcessType, Status

*"*PeopleCode Built-in Functions*,"* CreateProcessRequest

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

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

SetItemFolder, SetOutputOption, AddDistributionOption

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

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

AddDistributionOption

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 Values for Output Type and Format for more details and values.

Parameters

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

OutDest, OutDestFormat, OutDestType

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

RunStatus

ProcessRequest Class Properties

In this section, we discuss the ProcessRequest class properties. The properties are discussed in alphabetical order.

EmailAttachLog

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

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

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

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

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

RunControlID

LanguageCd

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 */
```

OutDest

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

OutDestType

PeopleTools PeopleBook: Peoplesoft Process Scheduler, "Managing PeopleSoft Process Scheduler"

OutDestFormat

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 Values for Output Type and Format for more details and values.

This property is read-write.

Example

```
&MYRQST.OutDestFormat = "RTF";
```

See Also

ProcessType, OutDestType

OutDestType

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 Values for Output Type and Format for more details and values.

This property is read-write.

Example

```
&MYRQST.OutDestType = "FILE";
```

See Also

ProcessType, RunControlID

PortalFolder

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

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

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

ProcessType, RunControlID

ProcessType

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

ProcessName, RunControlID

RunControlID

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

JobName, ProcessName, ProcessType

RunDateTime

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

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

PeopleTools PeopleBook: Peoplesoft Process Scheduler, "Using Process Monitor," Viewing the Status of Processes

RunRecurrence

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

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
4	Hold

RunStatus	*Description*
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

UpdateRunStatus

Status

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

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 22

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 *object* does 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

"Session Class"

PeopleTools PeopleBook: Peoplesoft Query, "Introduction to PeopleSoft Query"

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

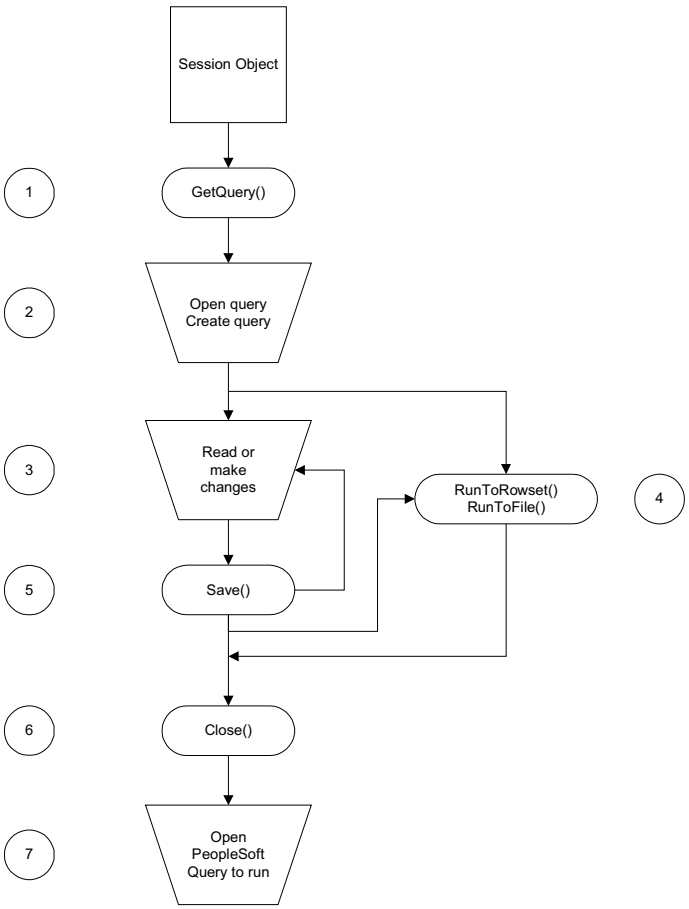
At runtime, there are certain things you want to do with a query, like creating one from scratch, updating the criteria for an existing query, running a query, and so on. The following is an overview of this process, and assumes the most common method to use the Query API. These steps are expanded in 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.

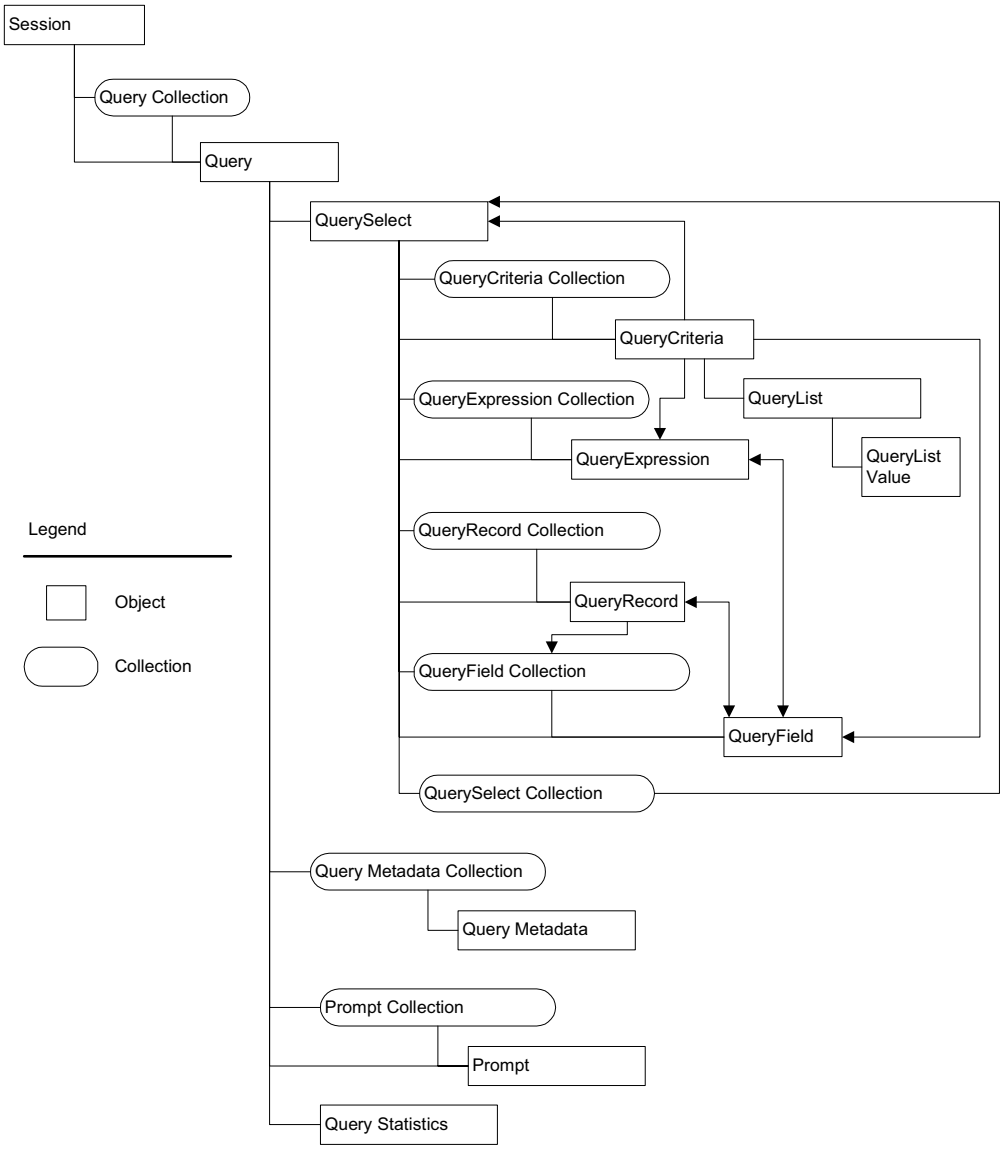
7. If you want, you can navigate to PeopleSoft Query Manager, Query Viewer or the Query Designer to run the query.



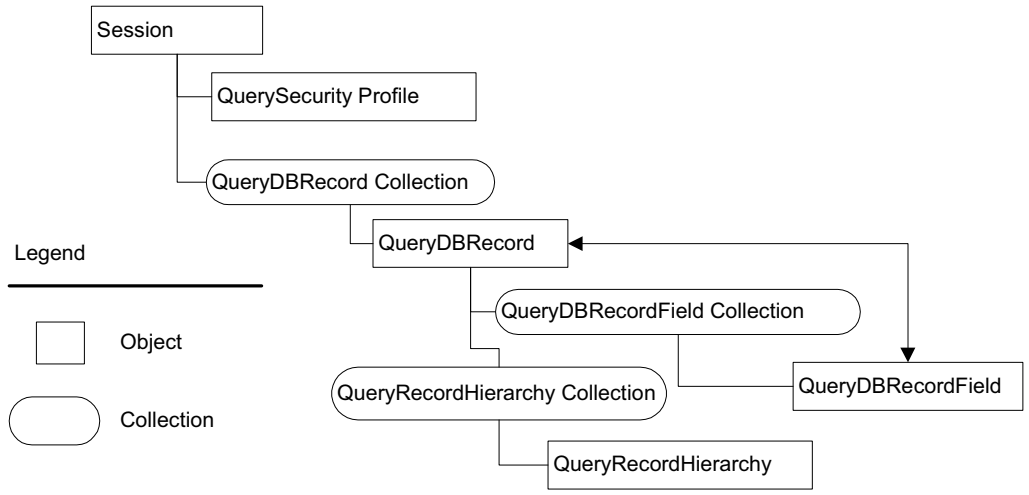
Life cycle of Query API

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	All the records available to be used as QueryRecords, and

QueryDBRecordFields	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

PeopleTools PeopleBooks: PeopleSoft Query, "Working With Criteria and Expressions“

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

PeopleTools PeopleBook: PeopleSoft Query, "Defining Selection Criteria"

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)
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

Operator

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

PeopleTools PeopleBook: Peoplesoft Query, "Appendix C: Query Monitor"

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 Also

RunToFile, RunToRowset

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

QuerySecurityProfile Class

PeopleTools PeopleBook: PeopleSoft Query, "Appendix B: PeopleSoft Query Security"

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

"Session Class," Error Handling

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

PeopleTools PeopleBook: Peoplesoft Query, "Defining Selection Criteria," Defining Having Criteria

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

"Session Class"

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

QueryDBRecord Collection, QueryDBRecord Class

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 = GetSession();
&MySession.Connect(1, "EXISTING", "", "", 0);
&MyList = &MySession.FindQueries();
```

See Also

FindQueriesDateRange, Query Collection, Open method

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

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

FindQueries, Query Collection, Open

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

FindQueries, FindQueriesDateRange, Open, Create

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

QuerySecurityProfile Class

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

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:

Constant Value	Numeric Value	Description
%Query_FindName	1	Search for records with the name matching the given pattern.
%Query_FindDescr	2	Search for records with the description matching the given pattern.
%Query_FindNameDescr	3	Search for records with either the name or the description matching the given pattern.

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

FindQueryDBRecords

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

QueryType — Specify the type of query to be searched for. You can specify either a constant or number value for this parameter. Values are:

Constant Value	Numeric Value	Description
%Query_ListQuery	1	Find queries of the type Query.
%Query_ListDBAgent	3	Find queries of the type DB Agent
%Query_ListRole	4	Find queries of the type Role

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:

Constant Value	*Numeric Value*	*Description*
%Query_FindName	1	Search for queries with the name matching the given pattern.
%Query_FindDescr	2	Search for queries with the description matching the given pattern.
%Query_FindNameDescr	3	Search for queries with either the name or the description matching the given pattern.

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

FindQueries, FindQueriesDateRange, SearchPublicQueries

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

QueryType — Specify the type of query to be searched for. You can specify either a constant or number value for this parameter. Values are:

Constant Value	Numeric Value	Description
%Query_ListQuery	1	Find queries of the type Query.
%Query_ListDBAgent	3	Find queries of the type DB Agent
%Query_ListRole	4	Find queries of the type Role

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:

Constant Value	Numeric Value	Description
%Query_FindName	1	Search for queries with the name matching the given pattern.
%Query_FindDescr	2	Search for queries with the description matching the given pattern.
%Query_FindNameDescr	3	Search for queries with either the name or the description matching the given pattern.

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

FindQueries, FindQueriesDateRange, SearchPrivateQueries

Query Collection

A query collection is returned from the following session methods:

- FindQueries
- FindQueriesDateRange
- SearchPrivateQueries
- SearchPublicQueries

See Also

FindQueries, FindQueriesDateRange, SearchPublicQueries, SearchPrivateQueries

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

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

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

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 Also

GetQuery, Query Collection

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

QuerySelect Collection

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

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

DeletePrompt, Prompts property, QueryPrompt Class

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

Open, Save, Create

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

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:

Value	Constant	Description
1	%Query_Query	Query
3	%Query_View	View
4	%Query_Role	Role
5	%Query_DBAgent	DBAgent

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

GetQuery, Close, Save

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

Save

DeletePrompt

Syntax

```
DeletePrompt(PromptNumber)
```

Description

The **DeletePrompt** method deletes a prompt from a query definition.

Parameters

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

AddPrompt, Prompts property, QueryPrompt Class

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

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

AddPrompt, Prompts

QueryPrompt Class

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

&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	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.

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

RunToFile, RunToRowset

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

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

GetQuery, Close, Save

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

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

Open, Close, Save

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

PeopleTools PeopleBook: Peoplesoft Application Designer, "Creating Style Sheet Definitions"

Parameters

&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:

Constant Value	*Numeric Value*	*Description*
%Query_PDF	2	The output is in PDF format.
%Query_HTML	5	The output is in HTML format.
%Query_XLS	8	The output for Excel in HTML format.
%Query_TXT	14	The output for txt in CSV format.

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

RunToRowset, PromptRecord

"PeopleCode Built-in Functions," Prompt

PeopleTools PeopleBook: 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

&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

RunToFile, **PromptRecord**

"PeopleCode Built-in Functions," Prompt

PeopleTools PeopleBook: 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

Open, Close

Query Class Properties

In the following section, we discuss each Query class property.

Approved

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.

Value	*Description*
A	Query has been approved by the query administrator.
M	Query has been modified.

This property is read-write.

ApproveOprId

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.

ApproveUserId

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

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

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.

CreateDtTm

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

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

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

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

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

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

This property returns the most recent date-time when the query was updated as a string.

This property is read-only.

LastUpdOprId

This property returns the User Id of the user who updated the query most recently as a string.

This property is read-only.

LongDescription

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

This property returns a metadata collection.

This property is read-only.

Example

```
&MetadataList = &MyQuery.Metadata;
```

See Also

Query Metadata Collection

Name

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

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

RunToFile, RunToRowset

Prompts

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

QueryPrompt Collection, RunTimePrompts

PromptRecord

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

RunToFile, RunToRowset

PublicPrivate

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:

Constant Value	*Numeric Value*	*Description*
%Query_Private	0	The query is a private query.
%Query_Public	1	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

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

QuerySelect Class

QueryStatistics

This property returns statistical information pertaining to the query's execution as a QueryStatistics object.

This property is read-only.

See Also

QueryStatistics Class

RunTimePrompts

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

QueryPrompt Collection, Prompts

SQL

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.

Type

This property specifies the type of query. This parameter takes either a constant or number value. Values are:

Constant Value	*Numeric Value*	*Description*
%Query_Query	1	Query of type Query
%Query_View	3	Query of type View
%Query_Role	4	Role
%Query_DBAgent	5	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 Also

QuerySelect Collection, QuerySelects

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

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

AddQueryOutputField, AddQueryRecord, AddQuerySelectedField

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

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

DeleteCriteria, Criteria

QueryCriteria Class

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

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

FindExpression, DeleteExpression, Expressions

QueryExpression Class

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

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.

Related Topics

DeleteHavingCriteria, HavingCriteria

QueryCriteria Class

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

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

QueryExpression Class, QueryField Class

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

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

AddAllFields

QueryField Class

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

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

QueryRecord Class

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

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

AddAllFields, DeleteField

QueryField Class

DeleteCriteria

Syntax

```
DeleteCriteria(index)
```

Description

The **DeleteCriteria** method deletes the criteria specified by *index* from the query definition.

Parameters

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

AddCriteria, Criteria

DeleteExpression

Syntax

```
DeleteExpression(index)
```

Description

The **DeleteExpression** method deletes the specified expression from the query definition.

Parameters

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

AddExpression, Expressions

QueryExpression Class

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

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

AddAllFields

QueryField Class

DeleteHavingCriteria

Syntax

```
DeleteHavingCriteria(index)
```

Description

The **DeleteHavingCriteria** method deletes the Having Criteria specified by *index* from the query definition.

Parameters

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

AddHavingCriteria, HavingCriteria

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

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

AddQueryRecord

QueryField Class

QuerySelect Properties

In this section, we discuss the QuerySelect properties. The properties are discussed in alphabetical order.

Criteria

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

QueryCriteria Collection

Distinct

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

This property returns a reference to a QueryExpression Collection.

This property is read-only.

See Also

QueryExpression Collection

HavingCriteria

This property returns a reference to a QueryCriteria collection populated with Having criteria.

This property is read-only.

See Also

QueryCriteria Collection

ParentSelectNum

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

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

QueryField Collection

QueryRecords

This property returns a reference to a QueryRecord Collection for the query.

This is property is read-only.

See Also

QueryRecord Collection

QuerySelectedFields

This property returns a reference to a QueryField Collection made up of QuerySelectedFields.

This is property is read-only.

See Also

QueryField Collection

QuerySelects

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

QuerySelect Collection

SelectNum

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

This property specifies what type of select statement. You can check for either a numeric or a constant value. The values are:

Constant Value	*Numeric Value*	*Description*
%Query_SelectMain	1	This is the Main Select of the query definition.
%Query_SubQuery	2	This is a subquery in the query definition.
%Query_Union	3	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 Also

QuerySelects

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

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

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

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

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 Also

QueryRecords

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

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

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

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 Also

QueryRecord Collection, AddQueryRecord, QueryRecord

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

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

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

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

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

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:

Constant Value	*Numeric Value*	*Description*
%Query_JoinNone	1	Query record isn't joined with any other record.
%Query_JoinHierarchy	2	Query record has a hierarchy join with another record.
%Query_JoinRelated	3	Query record has a lookup table join with another record.
%Query_JoinTree	4	Query record has a tree join with another record.

This property is read-write.

Name

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

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

QueryField Collection

RecordAlias

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 Also

QueryOutputFields, QuerySelectedFields, QueryFields

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

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

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

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

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 Also

AddQuerySelectedField, AddQueryOutputField, QueryField Collection

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

ExpressionName	Specify the name of the expression name that you want to add.

Returns

A reference to a QueryExpression object.

See Also

QueryExpression Class, Expressions property, DeleteExpression, AddExpression, AddTranslateField

AddTranslateField

Syntax

AddTranslateField(*FieldName*)

Description

Use the **AddTranslateField** method to add a Translate Effective Date Logic field for a translatable field.

Parameters

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

QueryExpression Class, Expressions property, DeleteExpression, AddExpression, AddTranslateExpression

QueryField Class Properties

In this section, we discuss the QueryField class properties. The properties are discussed in alphabetical order.

Aggregate

This property returns or sets the aggregate type for the query field.

This property takes either a constant or numeric value. The values are:

Constant Value	*Numeric Value*	*Description*
%Query_TtlNone	1	Query field has no total
%Query_TtlSum	2	Query field is for Sum
%Query_TtlCount	3	Query field is for Count
%Query_TtlMin	4	Query field is for Minimum
%Query_TtlMax	5	Query field is for Maximum
%Query_TtlAvg	6	Query field is for Average

This property is read-write.

ColumnNumber

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

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

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

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

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

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 "
Date	"DATE "
Time	"TIME "
Datetime	"DATE/TIME "
Image	"IMAGE "

This property is read-only.

HeadingText

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

This property returns or sets the heading type for the query field. This property takes either a numeric or constant value. The values are:

Constant Value	*Numeric Value*	*Description*
%Query_HdgNone	1	Query field has no heading.
%Query_HdgText	2	Query field has a text heading.
%Query_HdgRftShort	3	Query field uses the short RFT heading.
%Query_HdgRftLong	4	Query fields uses the long RFT heading.

This property is read-write.

HeadingUniqueFieldName

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

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

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

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

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*	*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

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

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

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

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

This property returns or sets the effective date logic for the QueryField.

The values are:

Constant Value	*Numeric Value*	*Description*
%Query_ExprCurDt	1	Current Date
%Query_ExprField	2	Field
%Query_Expression	3	Expression

This property is read-write.

TranslateExpression

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

QueryExpression Class

TranslateField

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

QueryField Class

TranslateOption

This property returns or sets the translate value for the QueryField. This property takes a numeric or constant value. The values are:

Constant Value	*Numeric Value*	*Description*
%Query_XlatNone	1	None
%Query_XlatShort	2	Short
%Query_XlatLong	3	Long

This property is read-write.

Type

This property returns the type of the field. You can use either a numeric or constant value. The values are:

Constant Value	*Numeric Value*	*Description*
%FieldType_Char	0	Character
%FieldType_LongChar	1	Long Character
%FieldType_Number	2	Number
%FieldType_SignedNumber	3	Signed number
%FieldType_Date	4	Date
%FieldType_Time	5	Time
%FieldType_DateTime	6	Datetime
%FieldType_Image	7	Image

This property is read-only.

QueryCriteria Collection

A QueryCriteria collection is returned from the Criteria QuerySelect class property.

See Also

Criteria

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

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

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

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 Also

AddCriteria, AddHavingCriteria, QueryCriteria Collection

Working With Criteria and Expressions

PeopleTools PeopleBook: PeopleSoft Query, "Defining Selection Criteria"

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

Expr1Expression, AddExpr1Field, Expr1Type

QueryExpression Class

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

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

Expr1Field, AddExpr1Expression, Expr1Type

QueryField Class

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

QueryExpression Class

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

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

QueryRecord Class, QueryField Class

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

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

QueryRecord Class, QueryField Class

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

QueryList Class

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

QuerySelect Class

QueryCriteria Class Properties

In this section, we discuss the QueryCriteria class properties. The properties are discussed in alphabetical order.

Expr1Expression

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

AddExpr1Expression

Expr1Field

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

QueryField Class

AddExpr1Field

Expr1Type

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:

Constant Value	*Numeric Value*	*Description*
%Query_ExprField	2	Field
%Query_Expression	3	Expression

This property is read-write.

Examples

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

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

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

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

AddExpr2Expression

Expr2Expression2

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

AddExpr2Expression

Expr2Field1

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

QueryField Class

AddExpr2Field1

Expr2Field2

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

QueryField Class, AddExpr2Field2

Expr2List

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

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

AddExpr2Subquery

Expr2Type

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.

See Also

Operator

Working With Criteria and Expressions

You can use either a constant or numeric value for this property. The values are:

Constant Value	*Numeric Value*	*Description*
%Query_ExprConstant	1	Constant
%Query_ExprField	2	Field
%Query_Exprression	3	Expression
%Query_ExprSubQuery	4	Subquery
%Query_ExprList	5	List
%Query_ExprCurDt	6	Current date
%Query_ExprTree	7	Tree
%Query_ExprBind	8	Bind
%Query_ExprBothConst	9	The criterion's operator is Between and both values on the right-hand side are constants. (Const-Const)
%Query_ExprConstFld	10	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)
%Query_ExprConstExpr	11	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)
%Query_ExprFieldConst	12	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)

Constant Value	Numeric Value	Description
%Query_ExprBothFld	13	The criterion's operator is Between and both values on the right-hand side are constants. (Field-Field)
%Query_ExprFldExpr	14	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)
%Query_ExprExprConst	15	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)
%Query_ExprExprFld	16	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)
%Query_ExprBothExpr	17	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

Expression2 Type	*Method or Property for Changing the Expression*	*Method or Property for Accessing the Expression*
Field-Field	AddExpr2Field(), AddExpr2Field()	Expr2Field, Expr2Field
Field-Expr	AddExpr2Field(), AddExpr2Expression()	Expr2Field, Expr2Expression
Expr-Const	AddExpr2Expression(), Expr2Constant	Expr2Expression, Expr2Constant
Expr-Field	AddExpr2Expression(), AddExpr2Field()	Expr2Expression, Expr2Field
Expr-Expr	AddExpr2Expression(), AddExpr2Expression()	Expr2Expression, Expr2Expression

This property is read-write.

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

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:

Constant Name	*Value*	*Description*
%Query_CombAnd	1	Logical And for non-having criteria
%Query_CombOr	2	Or for non-having criteria
%Query_CombNotUsed	3	No logical operator. Used for the first non-having criteria.
%Query_CombHaveAnd	4	Logical And for having criteria
%Query_CombHaveOr	5	Logical Or for having Criteria
%Query_CombHaveNotUsed	6	No logical operator. Used for the first having criteria.

This property is read-write.

LParenLvl

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

This property returns the name of the Query Criteria, as a string.

This property is read-only.

Negation

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

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.

See Also

Expr2Type

Working With Criteria and Expressions

You can use either a constant or numeric value for this property. The values are:

Constant Value	*Numeric Value*	*Description*
%Query_CondNone	1	None (used for initializing a new criteria.)
%Query_CondEqual	2	Criteria's left-hand side is equal to right-hand side (= operator)
%Query_CondNotEqual	3	Criteria's left-hand side is not equal to right-hand side (<> operator)
%Query_CondGreaterThan	4	Criteria's left-hand side is greater than right-hand side (> operator)
%Query_CondNotGreaterThan	5	Criteria's left-hand side is not greater than right-hand side (<= operator)
%Query_CondLessThan	6	Criteria's left-hand side is less than right-hand side (< operator)
%Query_CondNotLessThan	7	Criteria's left-hand side is not less than right-hand side (>= operator)
%Query_CondInList	8	Criteria's left-hand side is in the given list (IN operator)
%Query_CondNotInList	9	Criteria's left-hand side is not in the given list (Not IN operator)
%Query_CondBetween	10	Criteria's left-hand side is between the two values of right-hand side (BETWEEN operator)
%Query_CondNotBetween	11	Criteria's left-hand side is not between the two values of right-hand side (BETWEEN operator)
%Query_CondExists	12	Criteria's left-hand side is the output of the subquery of right-hand side (EXISTS operator)
%Query_CondNotExists	13	Criteria's left-hand side doesn't exist in the output of the subquery of right-hand side (NOT EXISTS operator)

Constant Value	*Numeric Value*	*Description*
%Query_CondLike	14	Criteria's left-hand side is like (wildcard search) the right-hand side (LIKE operation)
%Query_CondNotLike	15	Criteria's left-hand side is not like (wildcard search) the right-hand side (NOT LIKE operation)
%Query_CondNull	16	Criteria's left-hand side is NULL (NULL operation)
%Query_CondNotNull	17	Criteria's left-hand side is not NULL (IS NOT NULL operation)
%Query_CondInTree	18	Criteria's left-hand side is from a list of nodes in Tree (IN operation)
%Query_CondNotInTree	19	Criteria's left-hand side is not from a list of nodes in Tree (NOT IN operation)
%Query_CondEffDtLessEqual	20	Criteria's left-hand side is an Effective Date and is less than or equal to the date on the right-hand side (<= operation)
%Query_CondEffDtGreaterEqual	21	Criteria's left-hand side is an Effective Date and is greater than or equal to the date on the right-hand side (>= operation)
%Query_CondEffDtLess	22	Criteria's left-hand side is an Effective Date and is less than the date on the right-hand side (< operation)
%Query_CondEffDtGreater	23	Criteria's left-hand side is an Effective Date and is greater than the date on the right-hand side (> operation)
%Query_CondFirstEffDt	24	Criteria's left-hand side is the first effective date (Function MIN())
%Query_CondLastEffDt	25	Criteria's left-hand side is the last effective date (Function MAX())
%Query_CondInTreeJoin	26	Criteria's left-hand side is an In Tree Join.

This property is read-write.

R1ExprNum

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

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

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

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

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 Also

Expressions

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

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

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

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 Also

AddExpression, QueryExpression Collection, QueryCriteria Class Methods, Expr1Expression, Expr2Expression1

Working With Criteria and Expressions

QueryExpression Class Properties

In this section, we discuss the QueryExpression class properties. The properties are discussed in alphabetical order.

Aggregate

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

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

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

This property returns or sets the unique expression number, as a numeric value.

This property is read-write.

IsXlatExpression

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

This property returns or sets the length of the expression, as a number.

This property is read-write.

Name

This property returns the name of the expression, as a string.

This property is read-only.

OutputField

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

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

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

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

This property returns or sets the field type of the expression.

You can specify either a constant or a numeric value. The values are:

Constant Value	*Numeric Value*	*Description*
%FieldType_Char	0	Character
%FieldType_LongChar	1	Long Character
%FieldType_Number	2	Number
%FieldType_SignedNumber	3	Signed number
%FieldType_Date	4	Date
%FieldType_Time	5	Time
%FieldType_DateTime	6	Datetime
%FieldType_Image	7	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 Also

Expr2List, AddExpr2List

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

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

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

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 Also

QueryList Class Methods

QueryListValue Class Methods

None.

QueryListValue Class Properties

In this section, we discuss the QueryListValue class properties. These properties are discussed in alphabetical order.

IsPrompt

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

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 Also

RecordHierachy

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

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

Name	Specify the name of an existing QueryRecordHierarchy within the QueryRecordHierarchy collection. If you

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

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 Also

QueryRecordHierarchy Collection

QueryRecordHierarchy Class Properties

In this section, we discuss the QueryRecordHierarchy class properties. The properties are discussed in alphabetical order.

Description

This property returns a description of the record node as a string.

This property is read-only.

Level

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

This property returns the name of the record node as a string.

This property is read-only.

ParentFlag

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 Also

Metadata

Using Query Metadata

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

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

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

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 Also

Query Metadata Collection

Using Query Metadata

Query Metadata Class Properties

In this section, we discuss the Query Metadata class properties. The properties are discussed in alphabetical order.

Name

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

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 Also

QueryStatistics

QueryStatistics Methods

None

QueryStatistics Class Properties

In this section, we discuss the QueryStatistics properties. The properties are discussed in alphabetical order.

AvgExecTime

This property returns the average execution time for the query as a string.

This property is read-only.

AvgFetchTime

This property returns the average fetch time for the query as a string.

This property is read-only.

AvgNumRows

This property returns the average number of rows fetched for the query.

This property is read-only.

ExecCount

This property returns the total number of times the query has been executed.

This property is read-only.

LastExecDtTm

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 Also

GetQuerySecurityProfile

QuerySecurityProfile Class Methods

None

QuerySecurityProfile Class Properties

In this section, we discuss the QuerySecurityProfile properties. The properties are discussed in alphabetical order.

AllowAnyJoin

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

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

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

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

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

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

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

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

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

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

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

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

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

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

MaxUnapprovedRows

MaxInTreeCriteria

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

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

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

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

LimitUnapproved

QueryDBRecord Collection

A QueryDBRecord collection is returned from the FindQueryDBRecords session method.

See Also

FindQueryDBRecords

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

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

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

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 Also

QueryDBRecord Collection

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

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

QueryDBRecordFieldByName, QueryDBRecordField Class

QueryDBRecordFieldByName

Syntax

QueryDBRecordFieldByName(*Name*)

Description

The **QueryDBRecordFieldByName** method returns the QueryDBRecordField object with the name *Name*.

Parameters

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

QueryDBRecordFieldByIndex, QueryDBRecordField Class

QueryDBRecord Class Properties

In this section, we discuss the QueryDBRecord class properties. The properties are discussed in alphabetical order.

Description

This property returns the description of the QueryDBRecord as a string.

This property is read-only.

Name

This property returns the name of the QueryDBRecord as a string.

This property is read-only.

QueryDBRecordFields

This property returns a reference to a QueryDBRecordField Collection.

This property is read-only.

See Also

QueryDBRecordField Collection

RecordHierachy

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

QueryRecordHierarchy Collection

PeopleTools PeopleBook: PeopleSoft Query, "Setting up Query Security"

QueryDBRecordField Collection

A QueryDBRecordField collection is returned from the QueryDBRecordFields QueryDBRecord property.

See Also

QueryDBRecordFields

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

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

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

SortCriteria — Specify the sort order for the list. This parameter can take either a constant or numeric value. The values are:

Constant Value	Numeric Value	Description
%Query_SortNameAsc	1	Sort database fields in ascending order based on field name.

Constant Value	Numeric Value	Description
%Query_SortNameDesc	2	Sort database fields in descending order based on field name.
%Query_SoftFldNumAsc	3	Sort database fields in ascending order based on field number.
%Query_SortFldNumDesc	4	Sort database fields in descending order based on field number.

Returns

None.

QueryDBRecordField Collection Property

In this section, we discuss the Count property.

Count

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 Also

QueryDBRecord Collection

QueryDBRecordField Class Properties

In this section, we discuss the QueryDBRecordField class properties. The properties are discussed in alphabetical order.

Decimal

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

This property returns the long description of the field as a string.

This is property is read-only.

Flag

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

Value	*Description*
512	Edit Translate
4096	Date Range Edit
8192	Yes/No Table Edit
16384	Edit Table
262144	From Search Field
524288	Through Search Field
8388608	Use Default Label Flag
16777216	Default Search Field

This is a read-only property.

Format

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

This property returns the length of the field as a number.

This property is read-only.

LongName

This property returns the long name of the field as a string.

This property is read-only.

LookupTableName

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

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

This property returns the name of the field as a string.

This property is read-only.

Shortname

This property returns the short name of the field as a string.

This property is read-only.

Type

This property returns the type of the field. You can specify either a constant or a numeric value for this property. The values are:

Constant Value	*Numeric Value*	*Description*
%FieldType_Char	0	Character
%FieldType_LongChar	1	Long Character
%FieldType_Number	2	Number
%FieldType_SignedNumber	3	Signed number
%FieldType_Date	4	Date
%FieldType_Time	5	Time
%FieldType_DateTime	6	Datetime
%FieldType_Image	7	Image

This property is read-only.

QueryPrompt Collection

A QueryPrompt collection is returned from the Prompts and RuntimePrompts Query class properties.

See Also

Prompts, RunTimePrompts

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

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

Name	Specify the name of an existing QueryPrompt within the QueryPrompt collection. If you specify an invalid name

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

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 Also

QueryPrompt Collection, AddPrompt

PeopleTools PeopleBook: PeopleSoft Query, "Defining Selection Criteria," Defining Prompts

QueryPrompt Properties

In this section, we discuss the QueryPrompt properties. The properties are discussed in alphabetical order.

EditType

This property returns or sets the edit type for the field. This property takes a number value. The values are:

Value	*Description*
1	No table edit
2	Prompt table
3	Translate table
4	Yes/No

This property is read-write.

FieldDecimal

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

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
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

This property returns or sets the field length.

This property is read-write.

FieldName

This property returns or sets the field name used with the prompt.

This property is read-write.

FieldType

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:

Constant Value	*Numeric Value*	*Description*
%FieldType_Char	0	Character
%FieldType_LongChar	1	Long Character
%FieldType_Number	2	Number
%FieldType_SignedNumber	3	Signed number
%FieldType_Date	4	Date
%FieldType_Time	5	Time
%FieldType_DateTime	6	Datetime
%FieldType_Image	7	Image

This property is read-write.

HeadingText

This property returns or sets the heading text for the prompt field.

This property is read-write.

HeadingType

This property returns or sets the heading type for the query field. This property takes either a constant or numeric value. The values are:

Constant Value	*Numeric Value*	*Description*
%Query_HdgNone	1	Query field has no heading.
%Query_HdgText	2	Query field has a text heading.
%Query_HdgRftShort	3	Query field uses the short RFT heading.
%Query_HdgRftLong	4	Query fields uses the long RFT heading.

This property is read-write.

LangCount

This property returns the total count of the language records for the current prompt.

This property is read-only.

Name

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

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

This property returns or sets the prompt table name for the prompt field.

This property is read-write.

UniquePromptName

This property returns or sets the unique prompt name for the prompt field.

This property is read-write.

UseCount

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.RecordAlias, "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 23

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

PeopleTools PeopleBook: 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

Delete, SelectByKey, Insert, Update

"SQL Class"

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

CreateRecord

GetRecord

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

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

CopyChangedFieldsTo, CopyFieldsTo

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

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

CompareFields, CopyFieldsTo

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

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

CompareFields, CopyChangedFieldsTo, IsChanged

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

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 = "A";
&REC.KEYF2 = "X";
&REC.Delete();
```

See Also

SelectByKey, Insert, Update

"Rowset Class," DeleteRow

"SQL Class"

PeopleTools PeopleBook: Peoplesoft Globalization Technologies, "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 Also

SetEditTable

"Field Class," EditError, MessageNumber and MessageSetNumber

"Meta-SQL," %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

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

SetEditTable, IsEditError

"Field Class," EditError, MessageNumber and MessageSetNumber

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

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 **bold**.

```
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

"PeopleCode Built-in Functions," GetField

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

"SQL Class"

PeopleTools PeopleBook: Peoplesoft Globalization Technologies, "Using Related Language Tables"

SetDefault

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

Delete, SelectByKey, Update

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

SearchDefault

"Field Class," SearchClear

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

"SQL Class"

PeopleTools PeopleBook: Peoplesoft Globalization Technologies, "Using Related Language Tables"

"PeopleCode Built-in Functions," SQLExec

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

Delete, Insert, Update

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

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

Delete, Insert, Update, SelectByKey

"SQL Class"

PeopleTools PeopleBook: Globalization, "Chapter Name," Understanding 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 will *not* be 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

"Field Class," SetDefault

PeopleTools PeopleBook: 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 Also

ExecuteEdits

PeopleTools PeopleBook: 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

%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

ExecuteEdits, IsEditError

"Field Class," EditError, MessageNumber, MessageSetNumber

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.

See Also

"SQL Class"

PeopleTools PeopleBook: Peoplesoft Globalization Technologies, "Using Related Language Tables"

Parameters

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

Delete, SelectByKey, Insert

Record Class Properties

In this section, we discuss the Record class properties. The properties are listed in alphabetical order.

FieldCount

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

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

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

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

"Row Class," IsDeleted

IsEditError

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

ExecuteEdits, IsEditError

"Field Class," EditError, MessageNumber, MessageSetNumber

Name

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

"Meta-SQL," %Table

ParentRow

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

"PeopleCode Built-in Functions," CreateRecord

RelLangRecName

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 24

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

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Data Buffer Access"

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Build-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

GetRow

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

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

ParentRowset

"Record Class," CopyFieldsTo, CopyChangedFieldsTo

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

GetPriorEffRow

"Rowset Class," GetCurrEffRow, DeleteEnabled

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

GetNextEffRow

"Rowset Class," GetCurrEffRow, DeleteEnabled

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

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

"PeopleCode Built-in Functions," GetRecord, CreateRecord

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

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

scrollname

"PeopleCode Built-in Functions," GetRowset

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

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

"PeopleCode Built-in Functions," GetRowset

Row Class Properties

In this section, we discuss the Row class properties. The properties are discussed in alphabetical order.

ChildCount

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

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

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-in Functions," Using Scroll Areas and Scroll Bars, Using Standalone Rowsets

"Rowset Class," DeleteEnabled

IsChanged

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. **Note**. 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

This property returns True if the row has been deleted.

This property is read-only.

Example

```
&tmp = &ROW.IsDeleted;
```

IsEditError

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

"Record Class," ExecuteEdits, IsEditError

"Field Class," EditError, MessageNumber, MessageSetNumber

IsNew

This property is True if the row is a new (inserted) row.

This property is read-only.

Example

```
&tmp = &ROW.IsNew;
```

ParentRowset

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

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

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

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

This property returns True if the row is part of a grid and has been selected either by the user or programmatically.

An end-user can select or deselect all rows in a grid by clicking in the upper left-hand corner of a grid (0, 0). The first time an end-user clicks on this location, all rows are marked as **Selected**. Additional clicks will toggle this selection.

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

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

"Field Class," Style

PeopleTools PeopleBook: Peoplesoft Application Designer, "Creating Field Definitions"

Visible

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.

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 25

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

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-in Functions," Using Standalone Rowsets

PeopleTools PeopleBook: 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

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-in Functions", Using Standalone Rowsets

Rowset Class Built-in Functions

CreateRowset

GetLevel0

GetRowset

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 remove deleted rows  |;
   REM +------------------------------------------------------------------+;
   &rsIn.ClearDeletesChanges();

End-Function;
```

See Also

Flush, FlushRow

"Row Class," IsChanged, IsNew

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. **Note.** 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

&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

"Row Class," CopyTo

"PeopleCode Built-in Functions," CreateRowset

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Understanding PeopleCode Classes and Objects," 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

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

Flush, FlushRow, InsertRow

"Record Class," Insert

PeopleTools PeopleBook: 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 can't 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.

If the sort order is important, you should sort the rows yourself using the Sort method when using Fill.

Parameters

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

CopyTo, Select, FillAppend

"SQL Class"

"PeopleCode Built-in Functions," CreateRowset

PeopleTools PeopleBook: Peoplesoft Globalization Technologies, "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 can't 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

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

CopyTo, Select, Fill

"SQL Class"

"PeopleCode Built-in Functions," CreateRowset

PeopleTools PeopleBook: Peoplesoft Globalization Technologies, "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

FlushRow, DeleteRow, InsertRow

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

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

DeleteRow, Flush, InsertRow

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

DeleteEnabled, GetRow, GetNextEffRow, GetNewEffRow

"PeopleCode Built-in Functions," GetRow

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

GetRow, GetNextEffRow, GetCurrEffRow

"PeopleCode Built-in Functions," GetRow

"Rowset Class," DeleteEnabled

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

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

"PeopleCode Built-in Functions," GetRow

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

ShowAllRows, Visible

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

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

DeleteRow

"Record Class," Delete

PeopleTools PeopleBook: 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. **Note.** 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

CopyTo

"PeopleCode Built-in Functions," RemoteCall, TransferPage

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

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

SelectNew

"Record Class," SelectByKey

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

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

Select

"Record Class", SelectByKey

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

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

"Field Class," SetDefault

PeopleTools PeopleBook: 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

HideAllRows

"Row Class," Visible

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

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;
```

The second example sorts the rows on child rowset by primary and secondary key fields:

```
&RS1.Sort(SCROLL.EN_BOM_COMPS, EN_BOM_COMPS.SETID, "A",
EN_BOM_CMOPS.INV_ITEM_ID, "A");
```

See Also

Select, GetRowset

"PeopleCode Built-in Functions," GetRowset

Rowset Class Properties

In this section, we discuss the Rowset class properties. The properties are discussed in alphabetical order.

ActiveRowCount

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

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

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

"Field Class," DataAreaCollapsed

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

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

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

PeopleTools PeopleBook: Application Designer, "Chapter Name," Using Scroll Areas and Scroll Bars

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-in Functions," Using Standalone Rowsets

"Row Class," DeleteEnabled

"Rowset Class," InsertEnabled

EffDt

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

"PeopleCode Built-in Functions," CreateRowset

EffSeq

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

"PeopleCode Built-in Functions," CreateRowset

InsertEnabled

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

DeleteEnabled

PeopleTools PeopleBook: Application Designer, "Chapter Name," Using Scroll Areas and Scroll Bars

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-in Functions", Using Standalone Rowsets

IsEditError

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

"Record Class," ExecuteEdits, IsEditError

"Field Class," EditError, MessageNumber, MessageSetNumber

Level

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

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

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

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

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

This property returns the row that is being displayed at the top of the scroll (if any) for the rowset. This property can be used with ActiveX controls contained in a scroll. ActiveX controls aren't bound to page data, so the PSControlInit event runs within the context of the entire page, that is, at level 0.

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.

Example

The following is located in the PSControlInit event, and is used to make sure the ActiveX Chart control displays the correct data.

```
&Scroll = GetRowset(Record.MYSCROLL);
&NewTop = &Scroll.TopRowNumber;
If &NewTop <> &OldTop Then
     /* Update ActiveX Chart with data from row &Scroll.GetRow(&NewTop) */
     &OldTop = &NewTop;
End-If;
```

CHAPTER 26

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

PeopleTools PeopleBook: PeopleSoft Internet Architecture 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.

See Also

"PortalRegistry Classes," BuildSearchIndex

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=AB
SENCE_HISTORY1
```

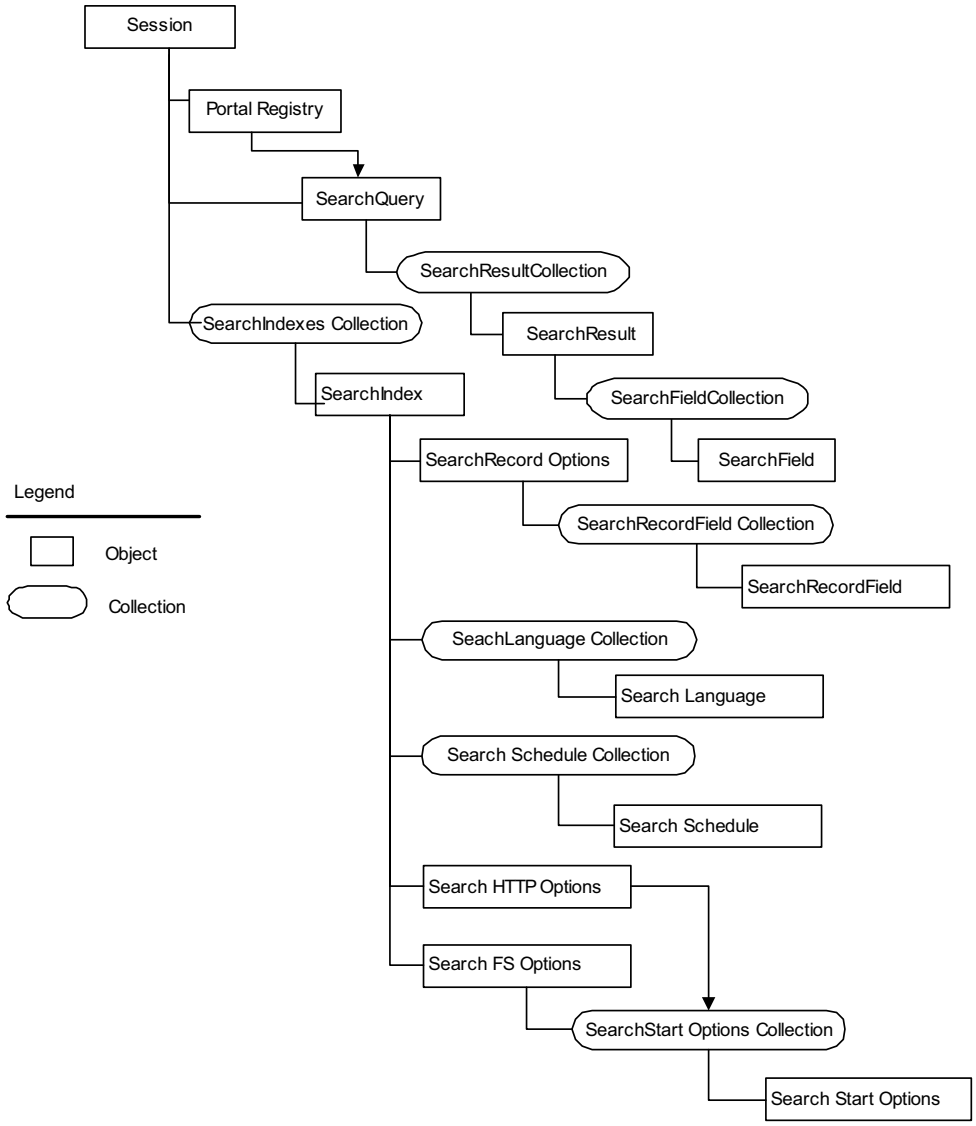
You can interrogate each field in the SearchField collection to find its name and value.

See Also

Key, Score, SearchFields

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.

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

"Session Class," Error Handling

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 doesn't 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

"Session Class"

"PortalRegistry Classes"

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

SearchIndexes Collection

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

"PortalRegistry Classes," GetSearchQuery

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

"ProcessRequest Class"

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

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

"Session Class," GetSearchQuery

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 Also

"PortalRegistry Classes," GetSearchQuery

"Session Class," GetSearchQuery

SearchQuery Class Method

In this section, we discuss the SearchQuery class method Execute.

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

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

SearchResult Collection

SearchQuery Class Properties

In this section, we discuss the SearchQuery class properties. The properties are discussed in alphabetical order.

HitCount

The **HitCount** property returns the total number of documents that actually match the query.

This property is read-only.

IndexName

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

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

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

The **ProcessedCount** property returns the total number of documents that were subjected to the search query.

This property is read-only.

QueryText

The **QueryText** property returns the query text to be submitted to the SearchQuery object.

This property is read-write.

SearchResult Collection

A SearchResult collection is returned from the Execute SearchQuery method.

See Also

Execute

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

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

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 Also

SearchResult Collection

SearchResult Class Properties

In this section, we discuss the SearchResult class properties. The properties are discussed in alphabetical order.

Key

The **Key** property returns the key for this SearchResult object.

This property is read-only.

Score

The **Score** property returns the score for this SearchResult object.

This property is read-only.

SearchFields

The **SearchFields** property returns a SearchField Collection.

This property is read-only.

See Also

SearchField Collection

SearchField Collection

A SearchField collection is returned by the SearchFields SearchResult property.

See Also

SearchFields

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

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

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 Also

SearchField Collection

SearchField Class Properties

In this section, we discuss the SearchField class properties. The properties are discussed in alphabetical order.

Name

The **Name** property returns the name of this SearchField object as a string.

This property is read-only.

Value

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 Also

GetSearchIndexes

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

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

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 Also

SearchIndexes Collection

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

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

This property returns a reference to a Search FS Options object, for FSYS type search indexes.

This property is read-only.

See Also

Search FS Options Class

HTTPOpts

This property returns a reference to a Search HTTP Options object, for HTTP type search indexes.

This property is read-only.

See Also

Search HTTP Options Class

Languages

This property returns a reference to the Search Language collection.

This property is read-only.

See Also

Search Language Collection

Location

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

This property returns the name of the SearchIndex, as a string.

This property is read-only.

RecOpts

This property returns a reference to a SearchRecord Options collection, for RECD type search indexes.

This property is read-only.

See Also

SearchRecord Options Class

Schedules

This property returns a reference to the Search Schedule collection.

This property is read-only.

See Also

Search Schedule Collection

Type

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)
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 Also

RecOpts

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

This property returns a reference to a SearchRecordField collection.

The SearchRecordField collection describes the records and fields that are indexed.

This property is read-only.

See Also

SearchRecordField Collection

Filter

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

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

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

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
<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

This property specifies the zone option for the index as a string. Values are:

Values	*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.

See Also

Fields

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

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

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

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

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 Also

SearchRecordField Collection

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

This property returns the field name for this SearchRecordField as a string.

This property is read-only.

IsAttachment

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

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

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

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 Also

Languages

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

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

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

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

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 Also

Search Language Collection

Search Language Class Properties

In this section, we discuss the Search Language class properties. The properties are discussed in alphabetical order.

LanguageCd

This property specifies the language code for the Search Language object as a string.

This property is read-only.

MapLanguageCd

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 Also

Schedules

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

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

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

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

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 Also

Search Schedule Collection

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

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

This property returns the Run Control ID associated with the schedule item.

This property is read-only.

RunRecurrence

This property specifies the name of the Recurrence Definition as a string. For example: "Daily Search Rebuild".

This property is read-write.

See Also

PeopleTools PeopleBook: Peoplesoft Process Scheduler, "Defining PeopleSoft Process Scheduler Support Information," Defining Recurrence Definitions

ServerName

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.

See Also

PeopleTools PeopleBook: Peoplesoft Process Scheduler, "Setting Server Definitions"

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 URL's. The properties available using this object represent values that must be set for spidering URL's.

See Also

HTTPOpts

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

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

This property limits spidering (indexing) to the specified domain. This property takes a string value.

This property is read-write.

GlobList

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

GlobListType

GlobListType

This property determines how the values specified with GlobList are used. Values are:

String 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

GlobList

LinkDepth

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

DomainLimit

MIMEList

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

MIMEListType, GlobList

MIMEListType

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

MIMEList, GlobListType

ProxyHost

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

This property specifies the port number of the HTTP proxy set in ProxyHost as a number.

This property is read-write.

StartOpts

This property returns a reference to a Search Start Options collection.

This property is read-only.

See Also

Search Start Options Collection

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 Also

FSOpts

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

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

GlobListType

GlobListType

This property determines how the values specified with GlobList are used. Values are:

String 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

GlobList

MIMEList

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

MIMEListType, GlobList

MIMEListType

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

MIMEList, GlobListType

StartOpts

This property returns a reference to a Search Start Options collection.

This property is read-only.

See Also

Search Start Options Collection

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 Also

StartOpts Search HTTP Options class property

StartOpts Search FS Options class property

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

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

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

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

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 Also

Search Start Options Collection

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

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

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

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

```
/*---------------------------------------------------------------------------
  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

```
&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("ContentProvider").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 27

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

"Component Interface Classes"

"PortalRegistry Classes"

"Query Classes"

"Search Classes"

"Tree Classes"

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

"Accessing a PSMessages Collection"

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.

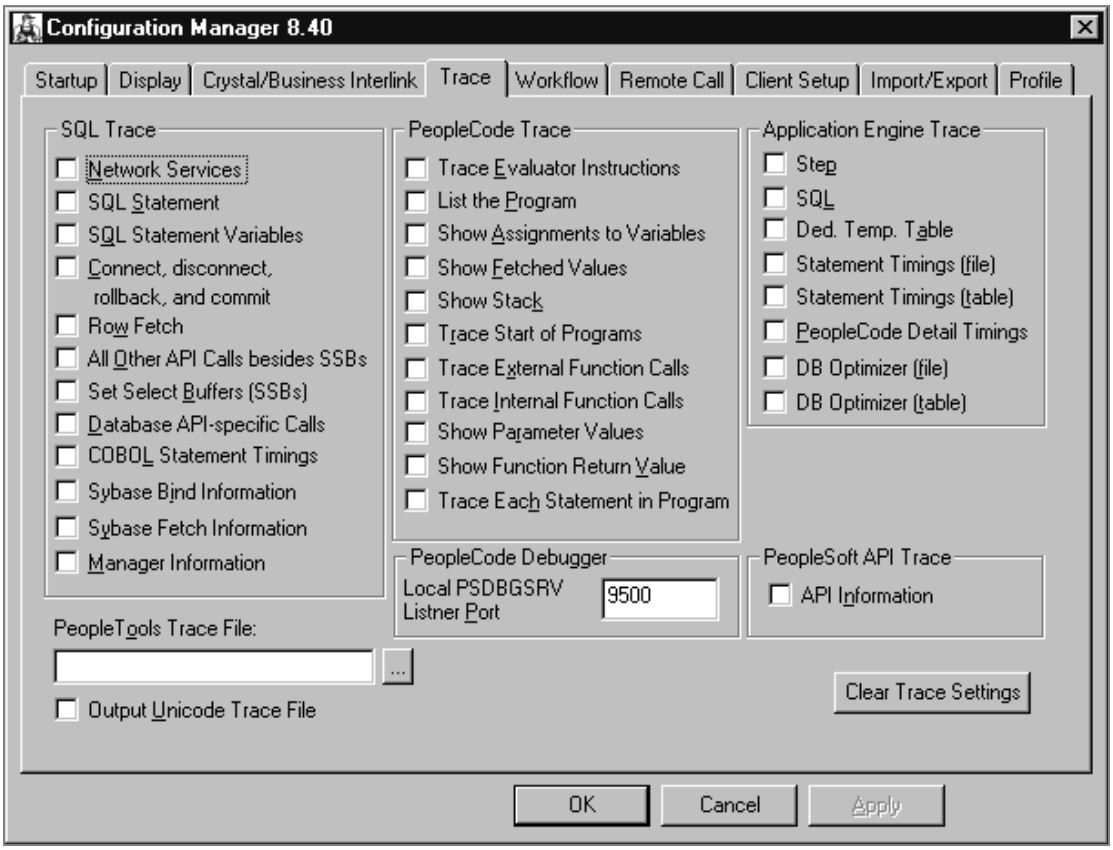
See Also

Controlling Windows Regional Settings

Regional Settings Class Properties

Trace Settings

PeopleSoft Configuration Manager is a PeopleTool that enables you to control the environment in which your PeopleTools session is running. The Trace tab in the Configuration Manager enables 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 options that appear on the Trace tab during your session.



Trace Tab of Configuration Manager

Not every option on the Trace tab is represented with a trace settings property. The initial value of a trace settings property depends on whether the session object used to instantiate the trace settings object was itself instantiated from within a component or from outside a PeopleSoft application.

- If the session object used to instantiate the Trace settings object was itself created within the context of a component (that is, from a running PeopleSoft application) the trace values currently in existence for that component are inherited by the trace settings object. For example, if the SQL trace property NetworkServices was set to True (the check box was selected) in PeopleSoft Configuration Manager before the session object was instantiated, the value for this property is initially set to True for the trace settings object.

- If the session object used to instantiate the trace settings object was itself *not* instantiated from a component (that is, from outside a PeopleSoft application), the initial value for all Boolean trace setting properties is False. The initial value for the PeopleTools trace file is the file that is set in PeopleSoft Configuration Manager for the workstation.

For both cases, when you make a change to the trace settings, these changes continue in the running environment even after the session is disconnected.

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

GetSession

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.

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.

Note. To use the existing session, use the %Session system variable to return a reference of a session object.

Parameters

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

"System Variables," %Session

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

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

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

PSMessages Collection Methods

PSMessagesMode

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:

Value	*Constant*	*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

This property returns a reference to a RegionalSettings collection object.

This property is read-only.

See Also

Regional Settings Class Properties

Repository

This property returns a reference to a Repository object.

See Also

PeopleTools PeopleBook: PeopleSoft Integration Tools and Utilities, "The PeopleSoft API Repository"

SuspendFormatting

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

This property returns a reference to a TraceSettings collection object.

This property is read-only.

See Also

Trace Setting Class Properties

WarningPending

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

ErrorPending

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

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

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

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

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

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

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

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:

Numeric 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.

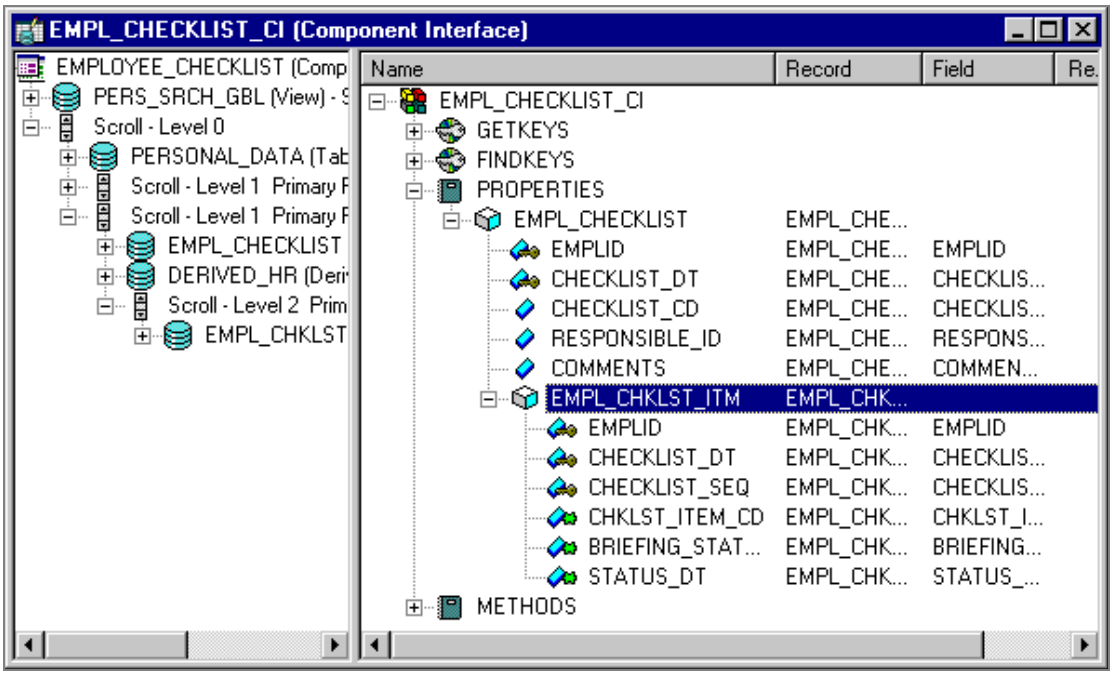
Source

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).[CollectionNa
me(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

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).[CollectionN
ame(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

Source

Regional Settings Class Properties

This section describes the regional settings class properties. The properties are discussed in alphabetical order.

ClientTimeZone

This property specifies the client time zone. This property takes a string value.

This property is read-write.

CurrencyFormat

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

This property specifies the currency symbol. This property takes a string value. The default value is "$".

This property is read-write.

DateFormat

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

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

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

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

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

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

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

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 Trace setting tab on PeopleSoft Configuration Manager.

This section describes the trace setting class properties. The properties are discussed in alphabetical order.

See Also

PeopleTools PeopleBook: Data Management, "Understanding Configuration Manager"

API

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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 28

SOAPDoc Class

This chapter provides an overview of SOAPDoc class and discusses the following topics:

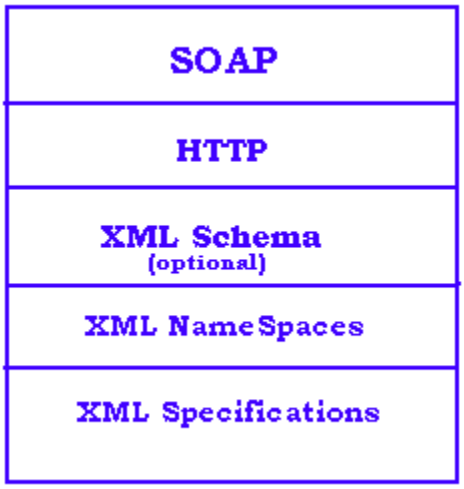
- SOAP message exchange model
- SOAP message document
- The SOAPDoc class
- SOAPDoc object creation
- The ValidateSOAPDoc method
- SOAPDoc objects and methods
- Data type of a SOAPDoc object
- Scope of a SOAPDoc object
- SOAPDoc built-in functions
- SOAPDoc class methods
- SOAPDoc class properties

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

PeopleTools PeopleBook: 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.

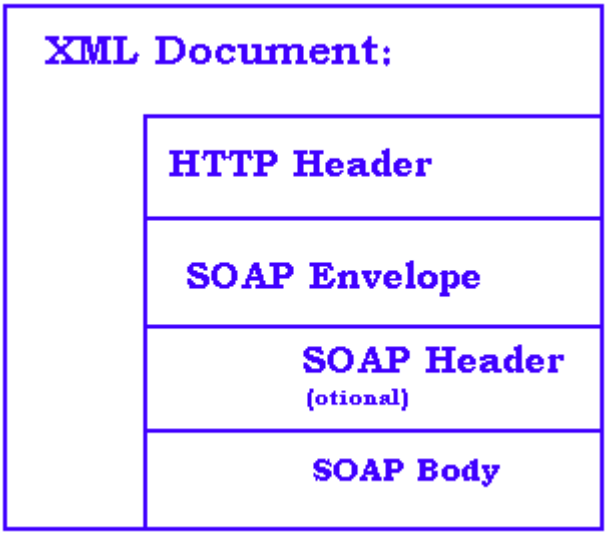
Processing the message requires validation of the mandatory sections of the SOAP XML document and understanding SOAP namespaces.

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**. 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, PeopleSoft doesn't support any PeopleCode functionality for the SOAP header settings or attributes. The header is ignored in inbound SOAP messages. Only SOAP messages using UFT8 encoding are supported. Alternative encoding styles are not supported. Therefore SOAPDoc objects using the **MustUnderstand=1** header attribute are considered an error when the SOAPDoc object is validated.

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

AddEnvelope

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

AddBody

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** or **InsertAttribute** XmlNode methods. 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

AddMethod, AddParm

"XmlDoc Class," AddAttribute, InsertAttribute

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

BodyNode, EnvelopeNode, MethodName, MethodNode, GetParmName, GetParmValue, ParmCount

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

AddFault, FaultCode, FaultString

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
- Does not use MustUnderstand = 1

See Also

ValidateSOAPDoc

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

"XmlDoc Class"

"PeopleCode Built-in Functions," SyncRequestXmlDoc, CreateXmlDoc, PublishXmlDoc

PeopleTools PeopleBook: 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

CreateSOAPDoc

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

"XmlDoc Class"

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

AddMethod, AddEnvelope

"XmlDoc Class," AddAttribute, InsertAttribute

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

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:

Value	Constant	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

AddBody

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

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.
200	MustUnderstand not supported	The MustUnderstand value and settings aren't supported by the implementation.
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("200", "MustUnderstand not supported");


/* ===> 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:200</faultcode>
  <faultstring>SOAP MustUnderstand not supported </faultstring>
  </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

See Also

FaultCode, FaultString

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

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

AddParm, MethodName, MethodNode

"XmlDoc Class," AddAttribute, InsertAttribute

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

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

AddMethod

GetParmName

Syntax

GetParmName(*Index*)

Description

Use the **GetParmName** method access the parameter names for a method.

Parameters

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

GetParmValue, ParmCount

GetParmValue

Syntax

GetParmValue(*Index*)

Description

Use the **GetParmValue** method to access the parameter values for a method.

Parameters

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

GetParmName, ParmCount

ValidateSOAPDoc

Syntax

```
ValidateSOAPDoc()
```

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.

Parameters

None.

Returns

This method returns either a number or a constant. The values are:

Value	Constant	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.
5	%SOAP_HasMustUnderstand	Unsupported MustUnderstand attribute in SOAPDoc.
6	%SOAP_InvalidXml	Invalid XML syntax in SOAPDoc.

Example

```
&Return = &MyDoc.ValidateSOAPDoc();
If &Return <> 0 Then
     /* do error processing */
End-if;
```

See Also

FaultCode, FaultString

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

"XmlDoc Class"

BodyNode

This property returns a reference to the body section of a SOAPDoc, as an XmlNode.

This property is read-only.

See Also

"XmlDoc Class"

EnvelopeNode

This property returns a reference to the envelope section of a SOAPDoc, as an XmlNode.

This property is read-only.

See Also

"XmlDoc Class"

FaultCode

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

FaultString, AddFault

FaultString

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

FaultCode, AddFault

MethodName

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

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

"XmlDoc Class"

ParmCount

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

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\\inputSOAP.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\\inputSoap.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 29

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

Understanding SQL Objects and PeopleSoft Application Engine Programs

"Record Class"

PeopleTools PeopleBook: 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

"Meta-SQL," %NoUppercase

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

Open

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Introducing the SQL Editor"

"PeopleCode Built-in Functions," FetchSQL, GetSQL , CreateSQL, StoreSQL

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

"Meta-SQL," %DTTM, %InsertValues, %KeyEqual, %KeyEqualNoEffDt, %List, %OldKeyEqual, %Table, %UpdatePairs

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

PeopleTools PeopleBook: 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

CreateSQL

DeleteSQL

FetchSQL

GetSQL

StoreSQL

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 */
     &ABSSQL.Execute(&ABS_HIST);
End-While;
&SQL.Close();
```

See Also

Open

"PeopleCode Built-in Functions," CreateSQL

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

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

Close, BulkMode

"PeopleCode Built-in Functions," CreateSQL

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

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.  */
End-While;
```

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

Open

"PeopleCode Built-in Functions," GetSQL, SQLExec

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

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

"PeopleCode Built-in Functions," GetSQL, CreateSQL

SQL Class Properties

In this section, we discuss the SQL class properties. The properties are discussed in alphabetical order.

BulkMode

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

Binding and Executing of SQL Statements

RowsAffected

IsOpen

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

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

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

Reusing a Cursor

RowsAffected

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

BulkMode

Status

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:

Constant	*Value*	*Description*
%SQLStatus_OK	0	No Errors
%SQLStatus_NotFound	1	Record Not Found
%SQLStatus_Duplicate	2	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

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
```

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

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 30

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

"PeopleCode Built-in Functions," TreeDetailInNode

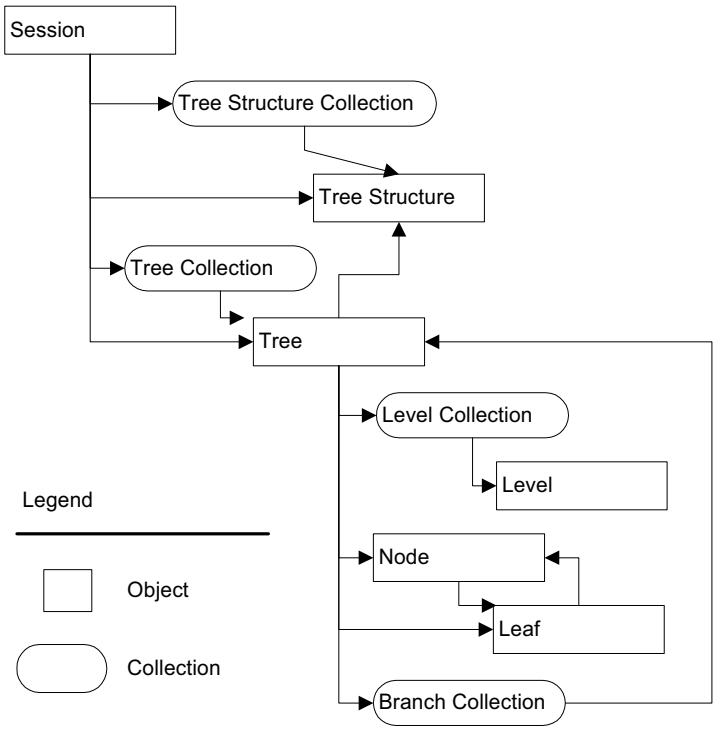
"Session Class"

PeopleTools PeopleBook: PeopleCode Developer's Guide, "Using Methods and Built-In Functions," Using the GenerateTree Function

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Introduction to Tree Manager"

Relationships Between Different Tree Classes

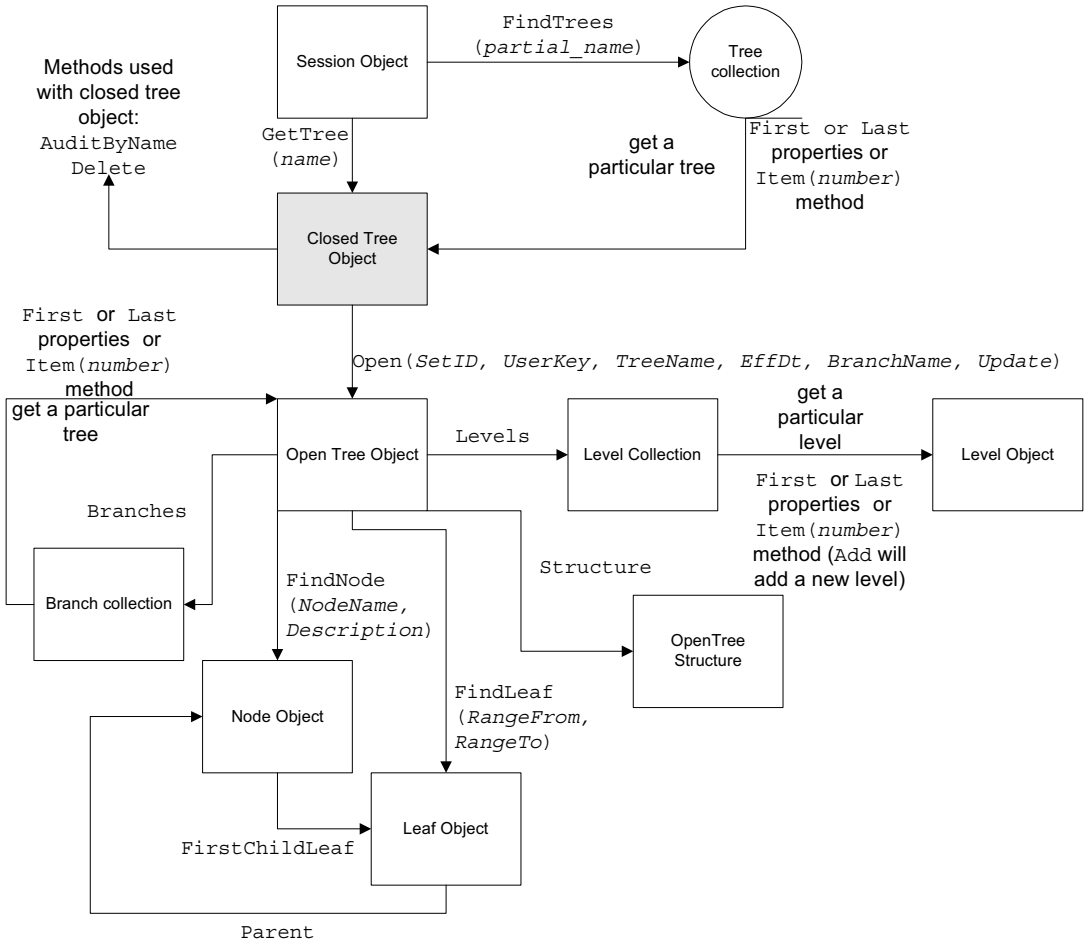
The following diagram shows the Tree classes and their relationships. Objects at the tip of the arrowhead are accessed or created from the object at the left of the arrowhead. Squares indicate objects. Ovals indicate collections.



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;
Local Boolean &RSLT;


&MYSESSION = GetSession();
&RSLT = &MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

If &RSLT 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

"Session Class," Error Handling

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

"PeopleCode Built-in Functions," None**,** All

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;
Local Boolean &RSLT;

&MYSESSION = GetSession();
&RSLT = &MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

If &RSLT 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 = GetSession();
&Session.CONNECT(1, "EXISTING", "", "", 0);

&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.

```
&Session = GetSession();
&Session.CONNECT(1, "EXISTING", "", "", 0);
```

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.

See Also

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Using TreeMover"

Session Class Methods

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

- 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

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;
Local Boolean &RSLT;

&MYSESSION = GetSession();
&RSLT = &MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

If &RSLT 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;
Local Boolean &RSLT;

&MYSESSION = GetSession();
&RSLT = &MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

If &RSLT 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 Also

Branches

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

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

Returns the number of branches for the branch collection object.

First

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

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

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 Also

FindLeaf, FirstChildLeaf, NextSib, PrevSib

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

Cut**,** PasteSib, PasteChild

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

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

InsertSib

"PeopleCode Built-in Functions," All, None

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

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

InsertDynSib

"PeopleCode Built-in Functions," All, None

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

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

MoveAsChildByName

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

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

MoveAsChild

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

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

MoveAsSibByRange

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

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

MoveAsSib

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

Cut leaf method, PasteChild, Cut node method

Leaf Class Properties

In this section, we discuss the Leaf class properties. The properties are discussed in alphabetical order.

DisplayLevelNumber

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

LevelNumber

Dynamic

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

PeopleTools PeopleBook: Tree Manager, "Creating Trees"

HasNextSib

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

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

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

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

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

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

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

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

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

This property returns the starting range, as a string, of the leaf.

This property is read-write.

RangeTo

This property returns the ending range, as a string, of the leaf.

This property is read-write.

TreeBranchName

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

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

This property returns the name of the tree as a string.

This property is read-only.

TreeSetId

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

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 Also

Levels

LevelCollection Methods

In this section, we discuss the LevelCollection 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

Number**,** AllValuesAudit

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

Returns the number of levels for the level collection object.

First

The **First** property returns a reference to the first level in the level collection executing the method *as an open object*.

Last

The **Last** property returns a reference to the last level in the level collection executing the method *as an open object*.

Next

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 Also

Item**,** First**,** Last**,** Next

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

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

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.

See Also

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Using Tree Manager," Modifying Tree Definitions

Description

This property returns the description of the level.

This property is read-write.

Name

This property returns the name of the level.

This property is read-write.

Number

This property returns the number of the level.

This property is read-write.

TreeBranchName

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

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

This property returns the name of the tree as a string.

This property is read-only.

TreeSetId

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

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 Also

FindNode**,** Parent **leaf property,** FirstChildNode, NextSib, PrevSib**,** Parent **node property**

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();
```

See Also

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Creating Trees," Working With Tree Branches

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

PasteSib, PasteChild, Cut 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

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

InsertDynChildLeaf

"PeopleCode Built-in Functions," All, None

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

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

"PeopleCode Built-in Functions," All, None

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

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

"PeopleCode Built-in Functions," All, None

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

InsertChildLeaf

"PeopleCode Built-in Functions," All, None

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

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

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

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

MoveAsChildByName

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

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

MoveAsChild

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

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

MoveAsSibByName

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

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

MoveAsSib

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

Cut leaf method, PasteSib, Cut 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

Cut leaf method, PasteChild, Cut 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

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();
```

See Also

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Creating Trees," Working With Tree Branches

Node Class Properties

In this section, we discuss the Node class properties. The properties are discussed in alphabetical order.

AllChildCount

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

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

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

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

This property returns the description of the node.

This property is read-only.

DisplayLevelNumber

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

LevelNumber

FirstChildLeaf

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

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

This property returns True if the node has immediate child leaves, False otherwise.

This property is read-only.

HasChildNodes

This property returns True if the node has immediate child nodes, False otherwise.

This property is read-only.

HasChildren

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

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

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

This property returns True if the node is branched, False otherwise.

This property is read-only.

IsChanged

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

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

This property returns True if the node has been deleted but the current tree hasn't been saved.

This property is read-only.

IsInserted

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

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

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

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

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

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

Rename

NextSib

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

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

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

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

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

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

This property returns the name of the tree as a string.

This property is read-only.

TreeSetId

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

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

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 = GetSession();
&MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

/* 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 Also

GetTree**,** First, Last, Next**,** Item

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

Open

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Using the Tree Audit/Repair Program**"**

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

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

Close

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Using the Tree Audit/Repair Program"

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

Open

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

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

Close, Open

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

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

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Creating Trees"

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

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", "");
```

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

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;
```

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

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

&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

DisplayLeaf, DisplayNode, FindPrevDisplayObject

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

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

&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

DisplayLeaf, DisplayNode, FindNextDisplayObject

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

Open

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

Open

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

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

NodeExists, FindLeaf

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

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;
```

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

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));
```

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

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

FindLeaf, FindNode, LeafExists

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

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

Close, Open

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.

See Also

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Using the Tree Audit/Repair Program"

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

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 optional Boolean value: True if the tree was saved with no errors, False if there is one or more errors.

Example

```
&MYTREE.SaveAs("", "", "DEPENDENT_BENEF2", "02-05-1999", "");
```

See Also

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Using the Tree Audit/Repair Program**"**

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

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 optional Boolean value: True if the tree was saved with no errors, False if there is one or more errors.

Example

```
&MYTREE.  SaveAsDraft("","","PERSONAL_EDIT2", "05-05-1997", "");
```

See Also

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Using the Tree Audit/Repair Program"

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.

This method returns an optional Boolean value: True if the tree was saved with no errors, False if there is 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.

See Also

Close, Open

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Using the Tree Audit/Repair Program"

Tree Class Properties

In this section, we discuss the Tree class properties. The properties are discussed in alphabetical order.

AllValues

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

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

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

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

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

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

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

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

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

FindNextDisplayObject, FindPrevDisplayObject

DisplayNode

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

FindNextDisplayObject, FindPrevDisplayObject

DuplicateLeaves

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

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

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

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

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

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

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

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

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.

See Also

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Using the Tree Audit/Repair Program**"**

IsWholeTree

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

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

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

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

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

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

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

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

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

LevelUse

Levels

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

LevelCollection Methods

LevelUse

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

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

Rename

NodeCount

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

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

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

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

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.

See Also

PeopleSoft Tree Manager

PS/nVision

PerformanceSelector

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.

See Also

PS/nVision

PerformanceSelectorOption

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.

See Also

PS/nVision

SetID

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

IndirectionMethod

Status

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

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

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

UserKeyValue**,** SetID

TotalNodeCount

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

NodeCount

UserKeyValue

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

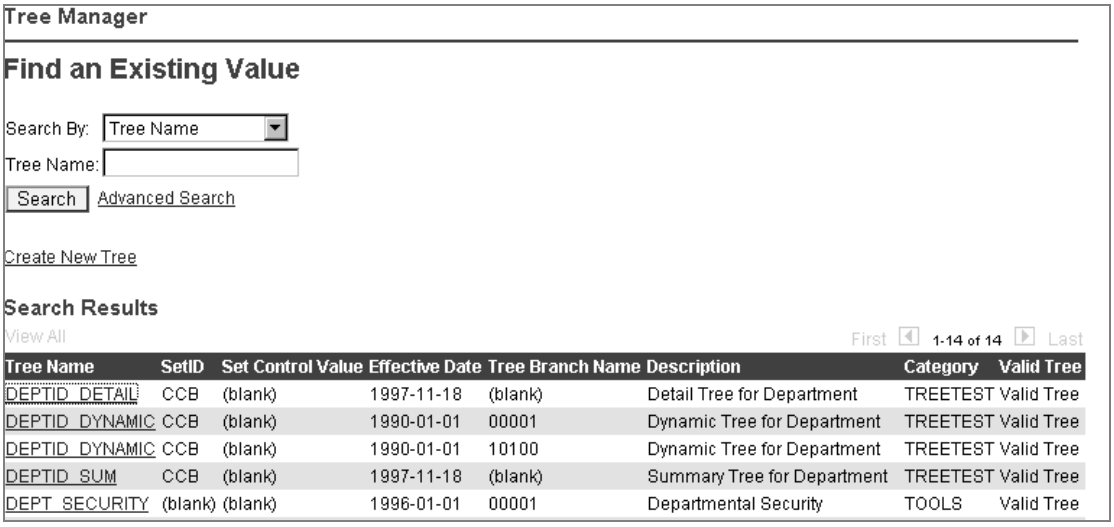
NodeRecord**,** IndirectionMethod

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 = GetSession();
&SESSION.Connect(1, "EXISTING", "", "", 0);
&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 Also

FindTree

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

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

KeyBranchName

Tree Collection Properties

In this section, we discuss the Tree collection properties. The properties are discussed in alphabetical order.

Count

Returns the number of trees for the tree collection object.

First

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

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

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 Also

GetTreeStructure**,** Structure

Tree Structure Collection

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

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

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

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

DetailRecord

DetailMenu

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

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

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

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

NodeMultiNavigate

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Appendix A: Setting Multi-Navigation Paths"

DetailPage

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

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.

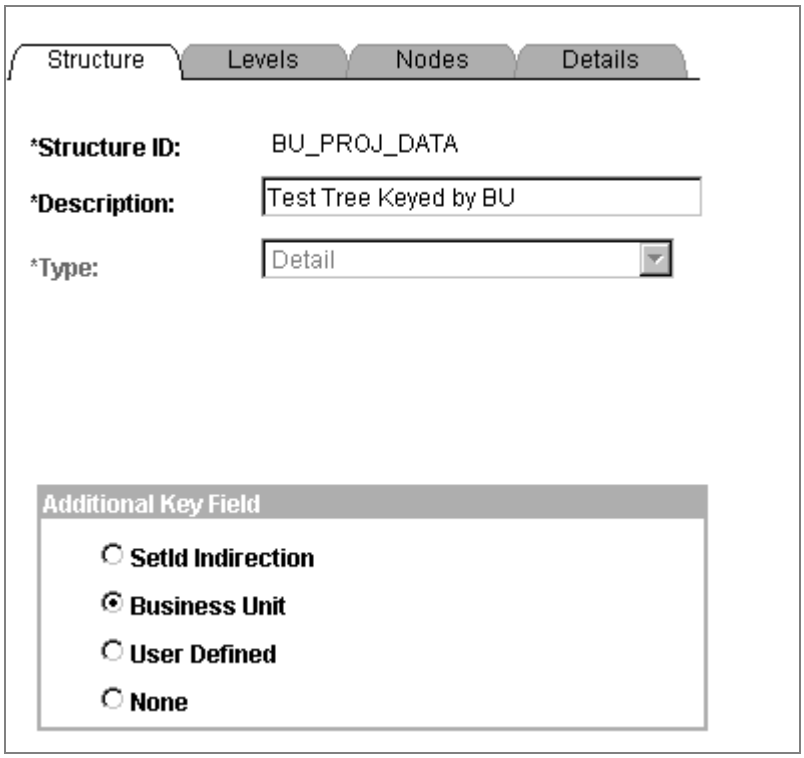
Example

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

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

IndirectionMethod**,** NodeRecord

KeyName

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 Boolean &RSLT;
Local string &NAME;

&MYSESSION = GetSession();
&RSLT = &MYSESSION.CONNECT(1, "EXISTING", "", "", 0);

If &RSLT 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

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

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

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

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

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

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

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

Rename

NodeComponent

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

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

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

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

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

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

DetailMultiNavigate

PeopleTools PeopleBook: Peoplesoft Tree Manager, "Appendix A: Setting Multi-Navigation Paths"

NodePage

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

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

IndirectionMethod

NodeUserKeyField

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

IndirectionMethod**,** UserKeyValue

SummarySetId

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

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

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

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

IndirectionMethod

Type

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.

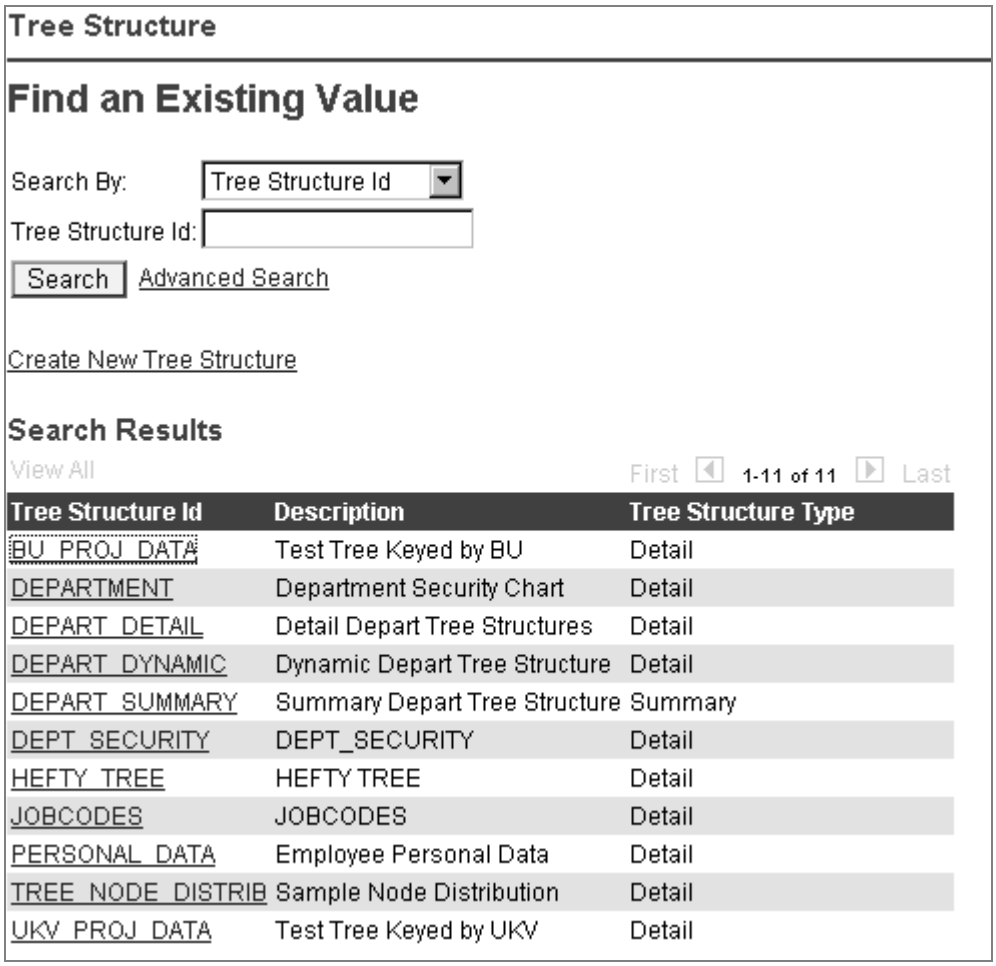
This property is read-write.

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 Also

FindTreeStructure

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

Returns the number of tree structures for the tree structure collection object.

First

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

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

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 31

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 the following is true:

- 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 **CreateDocument** method to specify the DTD, followed by the **CreateDocumentElement** method to associate the DTD with the root entity.

The following PeopleCode program:

```
&Doc = CreateXmlDoc("");

&DocType = &Doc.CreateDocument("Personal", "Personal.dtd");

&Root = &Doc.CreateDocumentElement("root", "", &DocType);
```

produces the following XML:

```
<?xml version="1.0"?>

<!DOCTYPE Personal PUBLIC "Personal.dtd">

<root>

</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

Internet Script Classes (iScript)"

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.

```
&MyNode = &MyDoc.FindNode("root/Request/Company/Location/User");
&Return = &MyNode.IsNull;

If Not(&Return) Then
     &Index = &MyNode.Index;
     &MyNode = &MyNode.NextSibling;

     &Return = &MyNode.IsNull;
     If &MyNode.IsNull Then
            &MyNewNode = InsertNode(&NewNode, &Index);
     End-if;
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 can't be used with a SOAPDocument. The following methods *cannot* be used with a SOAPDocument.

- AddCDataSection
- AddProcessInstruction
- CreateDocumentType
- InsertCDataSection
- InsertProcessInstruction

See Also

"SOAPDoc Class"

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

CancelPubHeaderXmlDoc

CancelPubXmlDoc

CancelSubXmlDoc

CreateSOAPDoc

CreateXmlDoc

GetMessageXmlDoc

GetNRXmlDoc

GetPubHeaderXmlDoc

GetPubXmlDoc

GetSubXmlDoc

GetSyncLogData

PublishXmlDoc

ReSubmitPubHeaderXmlDoc

ReSubmitPubXmlDoc

ReSubmitSubXmlDoc

ReValidateNRXmlDoc

SyncRequestXmlDoc

Transform

UpdateXmlDoc

XmlDoc Methods

The following are the methods provided by Peoplesoft for use with an XmlDoc object.

CopyRowset

Syntax

```
CopyRowset(&Rowset, 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 the rowset object.

The type of rowset used to populate an XmlDoc object must be a *structured* rowset, that is, based on some kind of structure. Rowsets created using an unstructured message are not allowed. However, the rowset could have been created using CreateRowset, from an Application Engine state record, and so on.

Parameters

&Rowset	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.
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.

See Also

CopyToRowset

"Rowset Class"

CopyPSFTMessage

Syntax

```
CopyPSFTMessage(targetDoc, srcPath, targetPath)
```

Description

Use the **CopyPSFTMessage** 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:

```
<messagename>

     <fieldtyes>
```

```
      ....

    </fieldtypes>

</messagename>
```

Parameters

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 File &FILE1;
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

CopyRowset, CopyToRowset

"Message Class"

"Array Class"

CopyToRowset

Syntax

```
CopyToRowset(&Rowset)
```

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 (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
```

If you wanted to include the information in the PO_DETAIL record, you would have to have *at least* the following record structure in your message:

```
PO_HEADER
     PO_INFO
            PO_DETAIL
```

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:

```
&msg = CreateMessage(Message.name);

&rs = &msg.GetRowset();

&doc = CreateXmlDoc();

&ret = &doc.CopyToRowset(&rs);
```

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>
```

Parameters

&Rowset	Specify the variable of an already instantiated rowset to copy data to.

Returns

This function always returns a True value, regardless of the success of the operation.

See Also

CopyRowset

"Message Class"

"Rowset Class"

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

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:

```
&Doc = CreateXmlDoc("");

&DocType = &Doc.CreateDocument("Personal", "Personal.dtd");

&Root = &Doc.CreateDocumentElement("root", "", &DocType);
```

produces the following XML:

```
<?xml version="1.0"?>

<!DOCTYPE Personal PUBLIC "Personal.dtd">

<root>

</root>
```

See Also

CreateDocumentType

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 eed 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 don't have to 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("Greeting", "", "SystemID.dtd");
```

The following may give you unexpected results if you were trying to specify a system ID, and not a public ID:

```
&DocType = &MyDoc.CreateDocumentType("Greeting", "SystemID.dtd");
```

This produces the XML:

```
<!DOCTYPE Personal PUBLIC "SystemID.dtd">
```

Parameters

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 string.

Example

The following example creates a document type declaration using the DTD named "personal".

```
&MyDoc = CreateXmlDoc("");
&DocType = CreateDocumentType("Personal", "", "personal.dtd");
&Root = &MyDoc.CreateDocumentElement("root", "", &DocType);
```

This produces the following XML:

```
<xml version="1.0"?>
<!DOCTYPE Personal SYSTEM "personal.dtd">
<root>
</root>
```

See Also

CreateDocumentElement

"PeopleCode Built-in Functions," CreateXmlDoc

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.

See Also

GenFormattedXmlString

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.

See Also

GenXmlString

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

TagName	Specify the tag name that you want to look for.

Returns

An array of XmlNode objects.

Example

```
&msg = CreateMessage(Message.name);
&rs = &msg.GetRowset();
&doc = CreateXmlDoc();
&ret = &doc.CopyToRowset(&rs);
&XmlNodes = &doc.GetElementsByTagName("Field1");

If &XmlNodes.Len = 0 Then
     /* no fields in record */
     /* do error processing */
Else
     /* do processing */
End-if;
```

See Also

"Array Class"

ParseXmlFromURL

Syntax

ParseXmlFromURL(*URL*)

Description

Use the **ParseXmlFromURL** method to convert the XML file located at *URL* 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

URL	Specify the URL, up to and including the file name, containing 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

```
&MyXml.ParseXmlFromURL("http://www.peoplesoft.com/xmlfile.xml");
```

See Also

ParseXmlString

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

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

```
&Ret = &MyDoc.ParseXmlString(&MyXmlString);

If &Ret Then
     /* do processing */
Else
     /* do error processing */
End-if;
```

See Also

ParseXmlFromURL

XmlDoc Property

The following is the XmlDoc property.

DocumentElement

This property returns a reference to the root element of the document as an XmlNode.

This property is read-only.

See Also

XmlNode

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 method creates a new node of type, append this new node to the child list, and return this newly created node.

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

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

A reference to the newly created attribute if successful. If not successful, the IsNull property is set to True.

Example

Here is a set of XML response code.

```
<?xml version="1.0"?>
<myroot>
   <postreqresponse>
      <candidate>
         <user>
            <location scenery="great" density="low" blank="eh?">
            </location>
         </user>
      </candidate>
   </postreqresponse>
</myroot>
```

Here's the PeopleCode that builds it.

```
&RootDoc = CreateXmlDoc("<?xml version='1.0'?><myroot></myroot>");

&XmlNode = &RootDoc.DocumentElement;

&postreqresponse = &XmlNode.addelement("postreqresponse");
&candidates = &postreqresponse.addelement("candidates");
&user = &candidates.addelement("user");
&location = &user.addelement("location");
&ret = &location.AddAttribute("scenery", "great");
&ret = &location.AddAttribute("density", "low");
&ret = &location.AddAttribute("blank", "eh?");

If &ret.IsNull Then
     /* do error processing */

Else
     &Value = &XmlNode.Genxmlstring();
```

```
      &MyFile = GetFile("XmlText.txt", "W");
      &MyFile.writeline(&Value);

End-if;
```

See Also

InsertAttribute, IsNull

AddAttributeNS

Syntax

```
AddAttributeNS(NamespaceURI, QualifiedName, 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

NamespaceURI	Specify the URI that contains the Namespaces for the XmlDoc, as a string.
QualifiedName	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

A reference to the newly created attribute if successful. If not successful, the IsNull property is set to True.

See Also

InsertAttributeNS, IsNull

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.

The new attribute is appended to the list of child nodes.

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 can't use this method with a SoapDoc object.

Parameters

Data	Specify the data to be included in the CData section as a string.

Returns

A reference to the newly created CData section.

See Also

InsertCDataSection

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

Text	Specify the text to be used as the comment, as a string.

Returns

A reference to the newly created comment.

Example

```
&Doc.AddComment("declarations for <head> & <body>");
```

See Also

InsertComment

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

TagName	Specify the tag for the new element that you are adding, as a string.

Returns

A reference to the newly created element.

See Also

InsertElement, GetElement, GetElementsByTagName

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

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:

```
&node = &root.AddElement("new");

&node1 = &node.AddEntityReference("MyURL");
```

Generates the following in the XML:

```
<new>&MyURL;</new>
```

See Also

InsertEntityReference

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

XmlNode	Specify the name of the node that you want to add.

Returns

A reference to the newly created node.

Example

```
&MyNode = &MyDoc.AddNode("Node Text");
```

See Also

InsertNode

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 can't use this method with a SoapDoc object.

Parameters

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.

See Also

InsertProcessInstruction

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

Data	Specify the data to be used as the text node, as a string.

Returns

A reference to the newly created text node.

Example

```
&MyText = &MyDoc.AddText("Some text here");
```

See Also

InsertText

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

Node	Specify the name of the node that you want to copy, as a string.

Returns

None.

Example

Suppose that your XmlDoc has the following structure:

```
<Node1>
     <Node11>
     </Node11>
</Node1>

<Node2>
     <Node22>
     </Node22>
</Node2>
```

The following PeopleCode copies Node2 and makes it a child of Node1.

```
&Node1.CopyNode(Node2);
```

The following is the new structure:

```
<Node1>
     <Node11>
     </Node11>
     <Node2>
            <Node22>
            </Node22>
```

```
      </Node2>
</Node1>
<Node2>
      <Node22>
      </Node22>
</Node2>
```

See Also

AddNode, InsertNode, GetChildNode, FindNode, FindNodes

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

Path	Specify the tag names up to and including the name of the node that you want returned, each separated by a slash (/).

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?">
            </location>
         </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

GetChildNode, FindNodes

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

Path	Specify the tag names up to and including the name of the node that you want returned, each separated by a slash (/).

Returns

An array of XmlNode objects. If there is no match for the specified path, an array with 0 elements is returned.

Example

```
&Ret = &MyDoc.FindNodes("root/Level1");
If &Ret.Len = 0 Then
     /* do error processing, no nodes returned */
Else
     /* do regular processing */
End-if;
```

See Also

GetChildNode, FindNode

"Array Class"

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

GenXmlString XmlDoc method

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.

See Also

AddCDataSection, InsertCDataSection, GetCDataValues

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

See Also

AddCDataSection, InsertCDataSection, GetCDataValue

"Array Class"

GetChildNode

Syntax

```
GetChildNode(index)
```

Description

Use the **GetChildNode** method to return the specified child node of an XmlNode.

Parameters

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

```
&MyNode = &doc.GetChildNode(11);
If &MyNode.IsNull Then
     /* Do error processing */
Else
     /* do regular processing */
End-if;
```

See Also

FindNode, FindNodes

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.

See Also

AddElement, InsertElement, GetElements, GetElementsByTagName

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

```
&Ret = &MyDoc.GetElements;
If &Ret.Len = 0 Then
     /* do error processing, no nodes returned */
Else
     /* do regular processing */
End-if;
```

See Also

AddElement, InsertElement, GetElement, GetElementsByTagName

"Array Class"

GetElementsByTagName

Syntax

GetElementsByTagName(*TagName*)

Description

Use the **GetElementsByTagName** method to return an array of XmlNode objects that match the specified tag name.

Parameters

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

```
&Ret = &MyDoc.GetElementsByTagname("MyTag");
If &Ret.Len = 0 Then
     /* do error processing, no nodes returned */
Else
     /* do regular processing */
End-if;
```

See Also

AddElement, InsertElement, GetElements,GetElement

"Array Class"

InsertAttribute

Syntax

```
InsertAttribute(Name, Value, Position)
```

Description

Use the **InsertAttribute** method to insert a new attribute in an XmlNode, at the location specified by *Position*.

Parameters

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.
Position	Specify where you want to insert the attribute, as a number.

Returns

A reference to the newly created attribute.

See Also

AddAttribute

InsertAttributeNS

Syntax

```
InsertAttributeNS(NameSpaceURI, QualifiedName, Position)
```

where *NamespaceURI* can have *one* of the following forms:

```
URL.URLname
```

OR a string URL, such as

```
http://www.peoplesoft.com/
```

Description

Use the **InsertAttributeNS** method to insert a new attribute in an XmlNode, at the location specified by *Position*.

Parameters

NamespaceURI	Specify the URI that contains the Namespaces for the XmlDoc.
QualifiedName	Specify the name of the attribute that you want to create, as a string.
Position	Specify where you want to insert the attribute, as a number.

Returns

A reference to the newly created attribute.

See Also

AddAttributeNS

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 can't use this method with a SoapDoc object.

Parameters

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.

See Also

AddCDataSection, GetCDataValue, GetCDataValues

InsertComment

Syntax

```
InsertComment(Data, Position)
```

Description

Use the **InsertComment** method to insert a comment in an XmlNode at the location specified by *Position*.

Parameters

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.

See Also

AddComment

InsertElement

Syntax

```
InsertElement(TagName, Position)
```

Description

Use the **InsertElement** method to insert an element in an XmlNode at the location specified by *Position*.

Parameters

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.

See Also

AddElement

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

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.

See Also

AddEntityReference

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

&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

```
&MyNode = &MyDoc.AddNode("Node Text");

&MyNode2 = &MyDoc.InsertNode(&MyNode, 11);
```

See Also

AddNode

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. You can't use this method with a SoapDoc object.

Parameters

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.

See Also

AddProcessInstruction

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

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.

See Also

AddText

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

Position | *Node*	Specify the child node that you want to delete, either using its position number or its name.

Returns

None.

See Also

AddNode, InsertNode, GetChildNode, FindNode, FindNodes, RemoveAllChildNode

RemoveAllChildNode

Syntax

```
RemoveAllChildNode()
```

Description

Use the **RemoveAllChildNode** method to remove all children nodes for an XmlNode.

Parameters

None.

Returns

None.

See Also

AddNode, InsertNode, GetChildNode, FindNode, FindNodes, RemoveChildNode

XmlNode Properties

The following are the XmlNode properties.

AttributeCount

This property returns the number of attributes for an XmlNode, as a number.

This property is read-only.

ChildNodeCount

This property returns the number of child nodes for an XmlNode, as a number.

This property is read-only.

Index

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

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

```
&MyNode = &doc.GetChildNode(11);
If &MyNode.IsNull Then
     /* Do error processing */
Else
     /* do regular processing */
End-if;
```

LocalName

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

Prefix

NamespaceURI

This property returns a reference to the namespace URI as a string.

This property is read-only.

NextSibling

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

IsNull

NodeName

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

This property returns the path of the XmlNode, from the root.

This property is read-only.

NodeType

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

Numeric Value	*Constant Value*	*Description*
9	%DocumentNode	Document node
10	%DocumentTypeNode	Document type node
12	%NotationNode	Notation node
13	%XmlDeclNode	XML DECL node

This property is read-only.

NodeValue

This property returns the value of the XmlNode, as a string.

This property is read-write.

ParentNode

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

IsNull

Prefix

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

LocalName

PreviousSibling

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

IsNull

CHAPTER 32

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

"PeopleCode Built-in Functions," PeopleCode Typographical Conventions

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**	

Class Name	*Functions, Methods and Properties*	*Returns*
	GetAESection(*applid ,ae_section* [, *effdt*])	AESection object
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
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*)	
	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**	

Class Name	*Functions, Methods and Properties*	*Returns*
	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
	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
	AddProcessingInstruction(*instruction*)	Number
	AddText(*text*)	Number

Class Name	*Functions, Methods and Properties*	*Returns*
	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()	
	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

Class Name	*Functions, Methods and Properties*	*Returns*
	RotationAngle	Number
	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**	
	XAxisLabelOrient	String
	XAxisStyle	String
	YAxisLabelOrient	String
	YAxisMax	Number
	YAxisMin	Number
	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**	

Class Name	*Functions, Methods and Properties*	*Returns*
	HasLegend	Boolean
	LegendMaxEntries	Number
	LegendPosition	String
	LegendStyle	String
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** Component Interface Class Methods and Properties	
Exception	**Functions**	
	CreateException(*message_set*, *message_num*, *default_txt [*, *subslist*])	Exception object
	Throw *Expression*	
	try	
	Protected StatementList	
	catch *QualifiedID &Id*	
	StatementList	
	end-try	
Exception	**Methods**	
	GetSubstitution(*Index*)	String
	Output()	Depends on context
	SetSubstitution(*Index*, *String*)	
	ToString()	String
Exception	**Properties**	
	Context	String, RO
	DefaultText	String
	MessageNumber	Number, RO

Class Name	*Functions, Methods and Properties*	*Returns*
	MessageSetNumber	Number, RO
	MessageSeverity	String, RO
	SubstitutionCount	Number, RO
Field	**Function**	
	GetField([*recname.fieldname*])	Field object
Field	**Methods**	
	GetAuxFlag(*FlagNumber*)	Boolean
	GetLongLabel(*LabelID*)	String
	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
	IsChanged	Boolean, RO
	IsDateRangeEdit	Boolean, RO
	IsDescKey	Boolean, RO
	IsDupKey	Boolean, RO
	IsEditTable	Boolean, RO
	IsEditXlat	Boolean, RO

Class Name	*Functions, Methods and Properties*	*Returns*
	IsFromSearchField	Boolean, RO
	IsInBuf	Boolean, RO
	IsKey	Boolean, RO
	IsListItem	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}
	LongTranslateValue	Any
	MessageNumber	Number, RO
	MessageSetNumber	Number, RO
	Name	String, RO
	OriginalValue	Depends on field
	ParentRecord	Record Object, RO
	PromptTableName	String, RO
	SeachDefault	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*])	
	FileExists(*filename* [, *pathtype*])	Boolean

Class Name	*Functions, Methods and Properties*	*Returns*
	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*)	
	WriteLine(*string*)	
	WriteRaw(*RawBinary*)	
	WriteRecord(*record*)	Boolean
	WriteRowset(*rowset* [, *Write_Data*])	Boolean
	WriteString(*string*)	
File	**Properties**	
	CurrentRecord	String, RO
	IgnoreInvalidId	Boolean
	IsError	Boolean, RO
	IsNewFileId	Boolean, RO
	IsOpen	Boolean, RO
	Name	String, RO
	RecTerminator	String
Grid	**Function**	
	GetGrid(PAGE.*pagename*, *gridname* [, *occursnumber*])	Grid object
Grid	**Methods**	
	*GetColumn(*columnname)*	Grid column object
	Properties	
	columnname	Grid column object
	Label	String

Class Name	*Functions, Methods and Properties*	*Returns*
GridColumn	**Properties**	
	Enabled	Boolean
	Label	String
	Name	String, RO
	Visible	Boolean
Internet Script	**See** Internet Script Classes Methods and Properties	
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
	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*])	

Class Name	*Functions, Methods and Properties*	*Returns*
	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*)	
	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
	SubName	String, RO
	SubscriptionProcessID	String, RO
OptEngine	**Functions**	
	CreateOptEngine(*probinst*, {**%Synch** | **%ASynch**} [, *Machine_Name*])	OptEngine object

Class Name	*Functions, Methods and Properties*	*Returns*
	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
	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** PortalRegistry Classes Methods and Properties.	
ProcessRequest	**Function**	
	CreateProcessRequest([*ProcessType*, *ProcessName*])	ProcessRequest object

Class Name	*Functions, Methods and Properties*	*Returns*
	GetNextProcessInstance([*Commit*])	Number
	SetupScheduleDefnItem(*ScheduleName, JobName*)	ProcessRequest object
ProcessRequest	**Method**	
	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
	EmailSubject	String
	EmailText	String
	EmailWebReport	Boolean
	JobName	String
	LanguageCd	String
	OutDest	String
	OutDestFormat	String
	OutDestType	String
	PortalFolder	String
	ProcessInstance	Number

Class Name	*Functions, Methods and Properties*	*Returns*
	ProcessName	String
	ProcessType	String
	RunControlID	String
	RunDateTime	DateTime
	RunLocation	String
	RunRecurrence	String
	RunStatus	Number
	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()	
	SetEditTable(*%PromptField*, RECORD.*recname*)	
	Update([*KeyRecord*])	Boolean
Record	**Properties**	
	FieldCount	Number, RO
	fieldname	Field Object, RO
	IsChanged	Boolean, RO
	IsDeleted	Boolean, RO

Class Name	*Functions, Methods and Properties*	*Returns*
	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
	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
	GetLevel0()	Rowset object
	GetRowset([SCROLL.*scrollname*])	Rowset object

Class Name	*Functions, Methods and Properties*	*Returns*
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
	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

Class Name	*Functions, Methods and Properties*	*Returns*
	ParentRow	Row Object, RO
	ParentRowSet	Rowset Object, RO
	RowCount	Number, RO
	TopRowNumber	Number, RO
Session	Many objects are instantiated from a session object. For more information, see Session Class Methods and Properties	
Session	**Function**	
	GetSession()	Session object
Session	**Methods**	
	Connect(*version*, {EXISTING | *ConnectID:Port}, UserID, Password, EXTAUTH*)	Boolean
	Disconnect()	Boolean
	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
	GetSearchIndexes()	SearchIndexes collection
	GetSearchQuery()	SearchQuery object

Class Name	*Functions, Methods and Properties*	*Returns*
	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**	
	CreateSOAPDoc()	SOAPDoc object
SOAPDoc	**Methods**	
	AddBody()	
	AddEnvelope(*SOAP_Schema*)	
	AddFault(*Fault_Code*, *Fault_String*)	
	AddMethod(*MethodName*, *IsResponse*)	
	AddParm(*ParmName*, *ParmValue*)	
	GetParmName(*Index*)	String
	GetParmValue(*Index*)	String
	ValidateSOAPDoc()	String
SOAPDoc	**Properties**	
	BodyNode	XmlNode object, RO
	EnvelopeNode	XmlNode object, RO
	FaultCode	Number, RO
	FaultString	String, RO

Class Name	*Functions, Methods and Properties*	*Returns*
	MethodName	String, RO
	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
	ReuseCursor	Boolean
	RowsAffected	Number, RO
	Status	Number, RO
	TraceName	String
	Value	String, RO
SyncServer	**System Variable**	
	%SyncServer	SyncServer object
SyncServer	**Properties**	
	ClientPlatform	String
	ConflictAlgorithm	String
	ConflictStatus	String
	SyncIDs	String

Class Name	*Functions, Methods and Properties*	*Returns*
	Synchronizing	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 Tree Classes Methods and Properties	
XmlDoc	**Functions**	
	CancelPubHeaderXmlDoc(*PubID, PubNode, ChannelName, VersionName*)	Boolean
	CancelPubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName, SubNode*)	Boolean
	CancelSubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName, SubscriptionName*)	Boolean
	GetMessageXmlDoc()	XmlDoc object
	GetNRXmlDoc(*NRID*, *NodeName*)	XmlDoc object
	GetPubHeaderXmlDoc(*PubID, PubNode, ChannelName, VersionName*)	XmlDoc object
	GetPubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName, SubNode*)	XmlDoc object
	GetSubXmlDoc(*PubID, PubNode, ChannelName, VersionName, MessageName*)	XmlDoc object

Class Name	*Functions, Methods and Properties*	*Returns*
	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, VersionName, MessageName, SubscriptionName*)	Boolean
	RevalidateXmlDoc	
	SyncRequestXmlDoc	
	Transform	
	UpdateXmlDoc(*&XmlDoc*, *PubID*, *PubNode*, *ChannelName*, *VersionName* [*, Message Name* [, *SubNode*]])	Boolean
XmlDoc	**Methods**	
	CopyRowset(*&Rowset*, *MessageName* [, *MessageVersion*])	Boolean
	CopyPSTFMessage(*targetDoc, srcPath, targetPath*)	Number
	CopyToRowset(*&Rowset*)	Boolean
	CreateDocumentElement(*TagName* [, *NamespaceURI*] [, *&DocType*])	XmlNode object
	CreateDocumentType(*Name*, *PublicID*, *SystemID*)	String
	GenXmlString()	String
	GenFormattedXmlString()	String
	GetElementsByTagName(*TagName*)	Array of XmlNode objects
	ParseXmlFromURL(*URL*)	Boolean
	ParseXmlString(*XmlString*)	Boolean
	Property	
	DocumentElement	XmlNode object, RO
XmlNode	**Methods**	
	AddAttribute(*Name*, *Value*)	XmlNode object
	AddAttributeNS(*NamespaceURI*, *QualifiedName*, *Value*)	XmlNode object

Class Name	*Functions, Methods and Properties*	*Returns*
	AddCDataSection(*Data*)	XmlNode object
	AddComment(*Text*)	XmlNode object
	AddElement(*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
	GetCDataValue()	String
	GetCDataValues()	Array of string
	GetChildNode(*index*)	XmlNode object
	GetElement()	XmlNode object
	GetElements()	Array of XmlNode objects
	GetElementsByTagName(*TagName*)	Array of XmlNode objects
	InsertAttribute(*Name*, *Value*, *Position)*	XmlNode object
	InsertAttributeNS(*NameSpaceURI*, *QualifiedName*, *Position*)	XmlNode object
	InsertCDataSection(*Data*, *Position*)	XmlNode object
	InsertComment(*Data*, *Position*)	XmlNode object
	InsertElement(*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
	RemoveChildNode({*Position* | *Node*})	
	RemoveAllChildNode()	
XmlNode	**Properties**	

Class Name	*Functions, Methods and Properties*	*Returns*
	AttributeCount	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 Component Interface Class Methods and Properties	
PortalRegisty classes	See PortalRegistry Classes Methods and Properties	
Query classes	See Query Classes Methods and Properties	
PSMessage Collection	**Methods**	
	DeleteAll()	Boolean
	DeleteItem(*number*)	Boolean
	First()	PSMessage object
	Item(*number*)	PSMessage object
	Next()	PSMessage object
PSMessage Collection	**Property**	
	Count	Number, RO

Objects	*Methods and Properties*	*Returns*
PSMessage	**Properties**	
	ExplainText	String, RO
	MessageNumber	Number, RO
	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 Summary of Repository Methods and Properties.	
TraceSettings	**Properties**	
	API	Boolean
	COBOLStmtTimings	Boolean
	ConDisRollbackCommit	Boolean
	DBSpecificCalls	Boolean
	ManagerInfo	Boolean
	NetworkServices	Boolean
	NonSSBs	Boolean
	OutputUNICODE	Boolean
	PCExtFcnCalls	Boolean
	PCFcnReturnValues	Boolean
	PCFetchedValues	Boolean

Objects	*Methods and Properties*	*Returns*
	PCIntFcnCalls	Boolean
	PCListProgram	Boolean
	PCParameterValues	Boolean
	PCProgramStatements	Boolean
	PCStack	Boolean
	PCStartOfPrograms	Boolean
	PCTraceProgram	Boolean
	PCVariableAssignments	Boolean
	RowFetch	Boolean
	SQLStatement	Boolean
	SQLStatementVariables	Boolean
	SSBs	Boolean
	SybBindInfo	Boolean
	SybFetchInfo	Boolean
	TraceFile	String
Tree, Tree Collection, Tree Structure, Tree Structure Collection	See Tree Classes Methods and Properties	

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

Class Name	*Functions, Methods and Properties*	*Returns*
Component Interface	**Methods**	
	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
	Save()	Boolean
Component Interface	**Properties**	
	ComponentName	String, RO
	CreateKeyInfoCollection	Component InterfacePropertyInfo collection object, RO
	FindKeyInfoCollection	Component InterfacePropertyInfo collection object, RO
	GetHistoryItems	Boolean
	GetKeyInfoCollection	Component InterfacePropertyInfo collection object, RO
	InteractiveMode	Boolean
	PropertyInfoCollection	Component InterfacePropertyInfo collection object, RO
	StopOnFirstError	Boolean
	Every user-defined property for a business component **definition** can be used like a property on the instantiated **object**.	Depends on property
CompIntfPropInfo Collection	**Methods**	

Class Name	*Functions, Methods and Properties*	*Returns*
	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
	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

Class Name	*Functions, Methods and Properties*	*Returns*
	Item(*number*)	data collection item
	ItemByKeys(*key_values*)	data collection item
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	**Method**	
	Execute(*Start*, *Size*)	SearchResult collection
	Properties	
	HitCount	Number, RO
	IndexName	String
	KnowledgeBase	String
	Language	String
	ProcessedCount	Number, RO
	QueryText	String
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
	Next()	SearchField

Class Name	*Methods and Properties*	*Returns*
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
	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

Class Name	*Methods and Properties*	*Returns*
	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**	
	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

Class Name	*Methods and Properties*	*Returns*
Search Schedule Collection	**Methods**	
	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
	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

Class Name	*Methods and Properties*	*Returns*
	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**	
	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

Class Name	*Methods and Properties*	*Returns*
	FindFolderByName(*Name*)	Folder object
	FindPgltByName(*PageletName*)	Pagelet object
	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
	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

Class Name	*Methods and Properties*	*Returns*
	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
	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**	
	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

Class Name	*Methods and Properties*	*Returns*
	Author	String, RO
	AuthorAccess	Boolean
	Authorized	Boolean
	CascadedPermissions	PermissionValue collection RO
	ContentRefs	ContentReference Collection, RO
	CreationDate	String, RO
	Description	String
	Folders	Folder collection, RO
	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
Folder Collection	**Properties**	
	Count	Number, RO
ContentReference	**Method**	
	Save()	Boolean

Class Name	*Methods and Properties*	*Returns*
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
	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

Class Name	*Methods and Properties*	*Returns*
	URL	String
	URLType	String
	UsageType	String
	ValidFrom	String
	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
	Next()	AttributeValue object
Attribute Collection	**Property**	
	Count	Number, RO
PermissionValue	**Properties**	
	Cascade	Boolean
	Name	String
PermissionValue Collection	**Methods**	

Class Name	*Methods and Properties*	*Returns*
	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
	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
	IsLayoutLocked	Boolean
	IsRenamable	Boolean
	Label	String
	LayoutBehavior	String
	Name	String, RO
	OwnerId	String

Class Name	*Methods and Properties*	*Returns*
	Product	String
	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
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

Class Name	*Methods and Properties*	*Returns*
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
	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

Class Name	*Methods and Properties*	*Returns*
	InsertItem(*Name*)	String
	ItemByName(*Name*)	String
	Next()	String
DynamicCategory Collection	**Property**	
	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
PageletCategory Collection	**Methods**	
	DeleteItem(*Name*)	Boolean
	First()	String
	InsertItem(*Name*, *Label*)	String
	ItemByName(*Name*)	String

Class Name	*Methods and Properties*	*Returns*
	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
	Description	String
	HelpID	String
	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
Pagelet Collection	**Methods**	
	DeleteItem(*Name*)	Boolean
	First()	Pagelet object
	InsertItem(*Name*, *Label*, *NodeName*, *URL*)	Pagelet object

Class Name	*Methods and Properties*	*Returns*
	ItemByName(*Name*)	Pagelet object
	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
	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**	

Class Name	*Methods and Properties*	*Returns*
	DeleteItem(*Name*)	Boolean
	First()	SelectedPagelet object
	InsertItem(*Name*)	SelectedPagelet object
	ItemByName(*Name*)	SelectedPagelet object
	Next()	SelectedPagelet object
SelectedPagelet Collection	**Property**	
	Count	Number, RO
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
	Item(*Number*)	Query object
	ItemByName(*Name*)	Query object
	Next()	Query object
Query Collection	**Property**	
	Count	Number, RO

Class Name	*Methods and Properties*	*Returns*
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
	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

Class Name	*Methods and Properties*	*Returns*
	Prompts	QueryPrompt collection, RO
	PromptRecord	Record object, RO
	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
	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

Class Name	*Methods and Properties*	*Returns*
	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
	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

Class Name	*Methods and Properties*	*Returns*
	Name	String, RO
	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
	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

Class Name	*Methods and Properties*	*Returns*
	RecordAlias	String, RO
	ShortName	String, RO
	TranslateEffDtLogic	String
	TranslateExpression	Expression object, RO
	TranslateField	QueryField, RO
	TranslateOption	String
	Type	String
QueryCriteria Collection	**Methods**	
	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
	Expr2Constant1	String
	Expr2Constant2	String

Class Name	*Methods and Properties*	*Returns*
	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
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

Class Name	*Methods and Properties*	*Returns*
	Name	String, RO
	OutputField	QueryField object
	RightExprFlag	Number
	SelectedField	QueryField object
	Text	String
	Type	Number
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
	Level	Number, RO
	Name	String, RO
	ParentFlag	Number, RO

Class Name	*Methods and Properties*	*Returns*
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
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

Class Name	*Methods and Properties*	*Returns*
	MaxInTreeCriteria	Boolean, RO
	MaxJoins	Boolean, RO
	MaxRowsToFetch	Boolean, RO
	MaxUnapprovedRows	Boolean, RO
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
	ItemByName(*Name*)	QueryDBRecordField object
	Next()	QueryDBRecordField object
	Sort(*SortCriteria*)	

Class Name	*Methods and Properties*	*Returns*
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
	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

Class Name	*Methods and Properties*	*Returns*
	LangCount	Number, RO
	Name	String, RO
	PromptRecordFieldName	String, RO
	PromptTable	String
	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
	Remove()	
	Save()	
Level Class	Create(*LevelName*, *LevelNumber*)	
	Delete()	

Class	*Method*	*Returns*
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*)	
	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

Class	Method	Returns
	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*)	
	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()	Boolean
	SaveAs(*SetID, UserKeyValue, TreeName, EffDt, BranchName*)	Boolean
	SaveAsDraft(*SetID, UserKeyValue, TreeName, EffDt, BranchName*)	Boolean
	SaveDraft()	Boolean
Tree Collection	Item(*SetId*, *UserKeyValue*, *TreeName*, *EffDt*, *BranchName*)	Tree object

Class	*Method*	*Returns*
Tree Structure	Close()	
	Copy(*FromStructId, ToStructId*)	
	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**	
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**

Properties	*Branch Coll*	*Leaf*	*Level*	*Node*	*Tree*	*Tree Structure*
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]
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**	
IsWholeTree					**RO**	

Properties	*Branch Coll*	*Leaf*	*Level*	*Node*	*Tree*	*Tree Structure*
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**
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**	

Properties	*Branch Coll*	*Leaf*	*Level*	*Node*	*Tree*	*Tree Structure*
NodePage						**X**
NodeRecord						**X**
NodeUserKeyField						**X**
Number			**X**			
Parent		**RO**		**RO**		
ParentLevel					**RO**	
ParentName					**RO**	
PerformanceMethod					**X**	
PerformanceSelector					**X**	
PerformanceSelectorOption					**X**	
PrevSib		**RO**		**RO**		
RangeFrom		**X**				
RangeTo		**X**				
SetID					**X**	
SilentMode					**X**	
State				**RO**		
Status					**X**	
Structure					**RO**	
StructureName					**X**	
SummarySetId						**X**
SummaryLevelNumber						**X**
SummaryTreeName						**X**
SummaryUserKeyValue						**X**
TreeBranchName		**RO**	**RO**	**RO**		
TreeEffDt		**RO**	**RO**	**RO**		
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.

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
	HTTPMethod	String, RO
	LogoutURL	String, RO
	PathInfo	String, RO
	PortalURI	String, RO
	Protocol	String, RO
	QueryString	String, RO

Class Name	*Methods and Properties*	*Returns*
	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
	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
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

Existing function name	*Use instead*
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
ScheduleProcess	CreateProcessRequest Function
ScrollFlush	Flush Rowset method
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	DataAreaCollapsed 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 properties
SetTraceSQL	If using API (Session object), use Trace Setting Class Properties properties
SortScroll	Sort Rowset method
TotalRowCount	RowCount Rowset property
Ungray	Enabled Field property
UnHide	Visible Field property
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
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

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 **bold** 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'

‘string’ | ‘lstring’ | ‘ustring’ | ‘float’ | ‘double’

CHAPTER 33

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

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.
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.

Field	Description
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
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

Field Name	Description	Remarks
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.

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:

```
/* Note: Keys of Tree are stored in &SetId,&TreeName variables.
Assume that application has Leaf selected with values stored in variables
&RangeFrom; &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;
```

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
&cPSTREEVIEWERWRK 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)

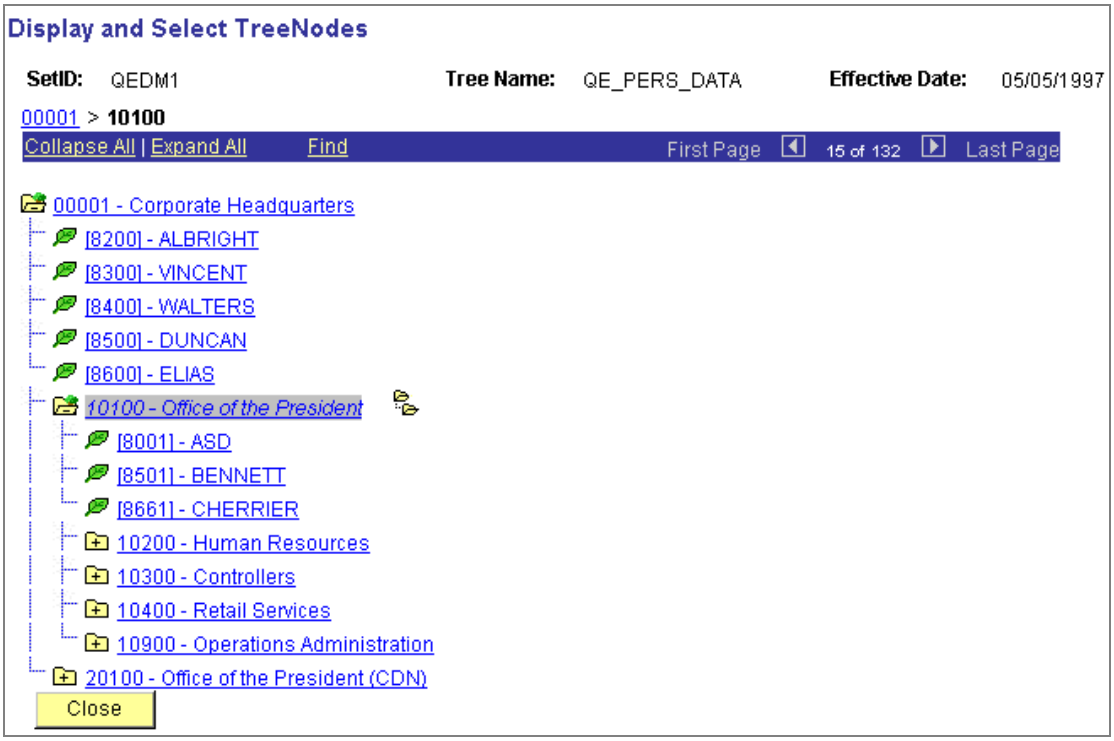
An example of an application that uses the Tree Viewer secondary page (PSTREEVIEWER), with the Multinodeselection flag set to "**N**", is the Using Copy/Delete Tree (PSTREEMAINT) component. Navigate to the component, select any tree and click the **View** button.

See Also

PeopleTools PeopleBook: Tree Manager, " Maintaining Trees," Maintain Trees

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)

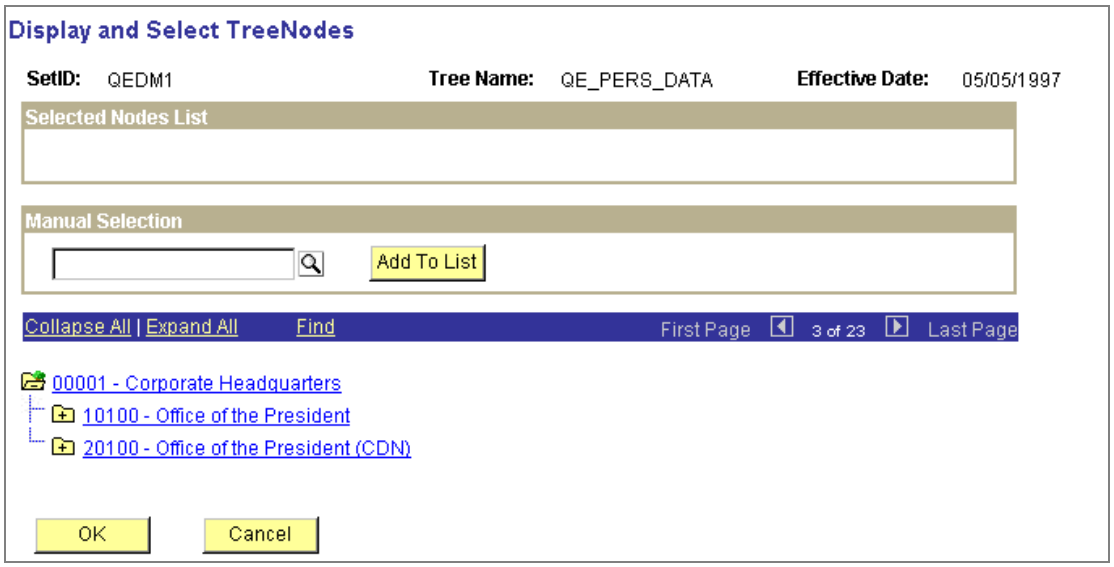
An example of an application that uses the PSTREEVIEWER secondary page, with the Multinodeselection flag set to "**Y**", is the Using Query Manager component. Navigate to the Query Manager component, select any query and click the Add Criteria Image. The Edit Criteria Properties page opens. Select *field* 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 Also

PeopleTools PeopleBook: Peoplesoft Query, "Defining Selection Criteria," Using Query Manager

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.

   The nodes you entered or selected from the list appear in the **Selected Nodes** drop-down list box.
3. Delete nodes from the list by clicking the Remove Node From List icon.
4. Display the selected node in the Tree by clicking the Find icon.
5. 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

The terms in this glossary are used among multiple Financials and Supply Chain Management applications.

Numbers

401(a)(17) Limits

The limitations on the earnings that may be included in the calculation of benefits under qualified U.S. pension plans.

1st Year Amount

In PeopleSoft Workforce Analytics, 1st Year Amount is an employee-level compensation amount, totaling the calculations for the first calendar year's worth of accounting periods, in a compensation scenario.

A

Abend

Abnormal End (to a process).

ABM (Activity-Based Management)

See PeopleSoft Activity-Based Management.

ABPS (Activity-Based Planning and Simulation)

See Activity-Based Planning and Simulation.

Absence

An absence occurs when an employee is not at work (absent) during a normally scheduled work period. Absences may be scheduled or non-scheduled, compensated or uncompensated, excused or unexcused. An absence may occur for a variety of reasons like illness, family emergency, civic obligations (e.g. Military duty or jury duty), or vacation.

Absence Entitlement

Element which defines the 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

Element which defines the conditions that must be met before a payee is entitled to take paid time off.

Accepted Exception

An exception that has been reviewed and validated (see Time Management).

Accommodations

Accommodations are efforts your organization is able to make for employees or applicants with disabilities, such as purchasing special equipment or making structural changes to a work environment.

Account Management

In PeopleSoft Demand Planning, a feature that enables you to divide a centrally held corporate forecast into multiple subsections for easier maintenance and management. These subsections are separate databases that can be distributed to account managers for use and updates, then rejoined with the main database at a later date.

Account

A code for recording and summarizing financial transactions as expenditures, revenues, assets, or liabilities balances. This is a delivered PeopleSoft ChartField, specific use of which is typically defined by the organization during implementation of PeopleSoft General Ledger.

Account Type

A name for one of the different kinds of accounts used in a PeopleSoft General Ledger, such as Asset, Liability, Equity, Revenue, and Expense.

Accounting Class

In PeopleSoft Enterprise Performance Management, an attribute that defines how the particular resource would be treated for generally accepted accounting practices. Inventory denotes whether a Resource will become part of a balance sheet account such as inventory or fixed assets, while Non-inventory denotes that the Resource will be treated as an expense of the period during which it occurs.

Accounting Date

The date that a transaction is recognized as opposed to the date the transaction actually occurred—the **Transaction Date** (although the two dates 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. In PeopleSoft Asset Management, the difference between accounting date and transaction date determines whether prior period depreciation must be calculated, and how much. Accounting Date must be later than or equal to Transaction 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 = credits). Accounting entries are created to record accruals, payments, payment cancellations, manual closures, project activities in general ledger, and so forth (depending on the application).

Accounting Entry Template

A user-defined table that controls the use of system-generated accounting lines in the posting processes.

Accounting Split

Method indicating how expenses are allocated or divided among one or more sets of accounting ChartFields.

Accredited Education

Education above the high school level completed in a U.S. college, university, or other educational institution that has been credited by one of the accrediting agencies or associations recognized by the Secretary, U.S. Department of Education.

Accrual

Any hours that employees accumulate for use at another time in the form of earned vacation time or sick leave, for example.

Accrual Basis Accounting

Accounting that records the impact of a business event as it occurs, regardless of whether the transaction affected cash.

Accrual Class Codes

Classes or categories of accruals.

Accrual Type

Defines an accrual such as annual leave or sick leave.

Accumulate Demand

In PeopleSoft Demand Planning, a transfer process function that adds demand quantities for an item to any quantities that already exist for the period.

Accumulator

Element which allows you to combine several elements. 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. See also Time Administration.

Accumulator [Global Payroll]

Element which provides a means for storing the cumulative values of defined items as they are processed. As you make payments, take deductions, and perform calculations, you'll use accumulators to track accumulated amounts, or balances. You can accumulate a single value over time or multiple values over time, as your requirements specify. 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

In PeopleSoft Deduction Manangement, a task that you perform to obtain information required to resolve a deduction.

Action and Conditions

A process that defines actions and conditions independently of one another and then combines them to create a complete rule (see Rule Creation).

Action Code

In PeopleSoft Engineering, a user-defined code associated with an event/action triggered by the implementation of an engineering change order (ECO). Actions could include analyzing an item's existing quantity on hand, scrapping existing inventory, or modifying current documentation.

In PeopleSoft Product Configurator, a 2-character code that identifies rule types. For example, ***FP*** is the action code for the Finalize Price rule, and ***CN*** is the action code for the Condition rule. The rules control the processing path for configured items.

Action List

An online list of customers who meet predefined credit management criteria. The list also includes appropriate procedures for each action and contact information for the customer.

Action Owner

In PeopleSoft Deduction Management, the individual assigned a task to obtain information to resolve a deduction.

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 paygroup to another—and a ***reason*** for that action. Action Reason is used by PeopleSoft Human Resources, PeopleSoft Benefits Administration, PeopleSoft Stock Administration, and the COBRA Administration feature of the Base Benefits business process.

Active Control

A target control requiring that the user validate the budget against the planning targets before submitting it. If the budget totals are not within the tolerance levels, the system indicates that the status is invalid and the user cannot submit their budget until the budget is modified and the amount is within the tolerance range of the planning target.

Activity

In PeopleSoft Receivables and Deduction Management, an action taken on an item, such as creating an item, unposting an item, or writing off an item.

In PeopleSoft Projects, the unit of work that provides a further breakdown of projects—usually into specific tasks. Resources are assigned directly to activities within a project, not directly to projects.

A self-contained task that is part of one or more business processes. Business process maps display the activities that make up the process. An activity consists of steps representing the pages the user needs to complete and events representing the workflow routings triggered by the user's actions.

In PeopleSoft Enterprise Warehouse, the work of an organization and the aggregation of actions used for Activity-Based Costing.

Activity Attributes

Activity Attributes provide pieces of activity information. For example: capacity and performance, cost drivers, cycle time and performance measures.

Activity-Based Costing (ABC)

A methodology that measures the cost and performance of activities, resources and cost objects, assigns resources to activities and activities to cost objects based on their use and recognizes the causal relationships of cost drivers to activities.

Activity-Based Management (ABM)

See PeopleSoft Activity-Based Management (ABM).

Activity-Based Planning and Simulation (ABPS)

ABPS, a feature of PeopleSoft Activity-Based Management, calculates resource demands, new rates, costs, and activity volumes based on demand forecasts. It converts the new

resource demands into new cost requirements at the General Ledger item level to feed as input for budgeting.

Activity Driver

An Activity Driver indicates the amount of demand there is for a particular activity and it is used to assign cost to cost objects. In some instances, an activity driver may represent the yield of an activity.

Activity Fragmentation

The part of the Employee Profile feature that provides information about the number of employees that is involved in completing a particular activity on a full or part-time basis.

Activity ID

A unique 15-character alphanumeric identifier given to each activity within a project. Activity IDs need only be unique within a single project.

Activity List

In PeopleSoft Pension Administration, a checklist used to monitor pension-related activities.

Activity Type

A user-definable identifier for grouping activities.

Activity Type

Also known as Activity Code. A categorization of work effort. Typically work effort is categorized as productive or non-productive; Repair, Maintenance, Enhancement, or Improvement; or Development or Construction. Activity type is usually required to support cost accounting or financial accounting (recording) functions. It may also be required to support some organizational administration requirements such as organizational productivity goals, or employee performance measurement. In some companies, activity type is inferred from job function, work group affiliation, or organization.

Activity Use

An attribute used to describe the behavior of an Activity as defined within PeopleSoft Enterprise Performance Management. A Primary Activity is an activity that is performed for the purpose of directly generating revenue within the course of business. A Secondary Activity is generally performed in direct support of a Primary Activity such as activities related to human resources or MIS.

Actual Base Hours

This defines the number of hours that an employee is expected to work within a given period under analysis within PeopleSoft Enterprise Performance Management. Hours worked in excess of Actual Base Hours are generally considered overtime, while hours worked less than Actual Base Hours would illustrate that the employee is working part-time.

Actual Contribution Percentage (ACP)

The amount of an employee's after-tax or employer matching contributions made in a Section 401(m) plan on behalf of highly compensated plan participants, divided by the employee's annual compensation, or an amount determined in the same manner with respect to non-highly compensated employees. The Base Benefits business process is set up to perform ACP nondiscrimination tests for Section 401(m) plans. See Nondiscrimination Tests and Highly Compensated Employee.

Actual Date

Calendar date in which a punch occurred (see Time Reporting).

Actual Deferral Percentage (ADP)

The amount of salary reduction contributions made by an employee to a Section 401(k) plan for a year, divided by the employee's total compensation for that year. The Base Benefits business process is set up to perform ADP nondiscrimination tests for Section 401(k) plans. See Nondiscrimination Tests and Highly Compensated Employee.

Actual Demand

In PeopleSoft Demand Planning, an **Array** of demand by historical period imported from an external system. The demand figures are determined by imported values and typically include shipments, orders booked, orders booked by requested ship date, or shipments.

Actual Rates

An Actual Rate is the rate that your business currently uses for its business practice.

Actuarial Assumptions

Any assumptions used to calculate an equivalent benefit for an optional form of payment or an alternative retirement date.

Actuarial Valuation

A comparison of a pension plan's assets and liabilities.

Actuarial Valuation Extract

A PeopleSoft Pension Administration data extract containing data that a plan actuary needs in order to determine the plan’s assets and liabilities.

Address Type

A high-level address classification that identifies addresses associated with a **Material Issue**. Examples include Ship To Address, Bill To Address, and Ship Notification Address.

Adjusted

In the Enterprise Planning and Simulation forecasting process, in addition to versions of the statistical forecast, there is an adjusted version of the forecast. Managers create this version by reviewing the forecasts and entering adjustments that cannot be inferred statistically. For example, there may be a promotional campaign next quarter that is expected to boost volume for certain products over several weeks.

Adjusted Demand

In PeopleSoft Demand Planning, an **Array** of demand after adjustments have been made to the actual demand values. The adjusted figures may include both manual and system-generated changes, such as demand filtering and depromotion. The system uses adjusted demand rather than actual demand in the Forecasting Reset process and in the recalculation of model components during period-end processing.

Adjusted Forecast

In PeopleSoft Demand Planning, a **Statistical Forecast** that has been adjusted using management overrides, proration, or summarization.

Adjustment

See **Bill Adjustment** or **Inventory Adjustment**.

Adjustment Voucher

A PeopleSoft Payables voucher that enables you to apply an adjustment to an existing voucher or to relate one voucher to another.

Advice

The Form that employees who choose direct payroll deposit receive in lieu of a check.

Affiliate

A control person of a corporation. Generally, an officer, director, or major shareholder that has the ability to influence the corporate management decisions.

After-tax Deductions

Deductions that reduce net pay. These deductions are subtracted from gross pay after taxes have been taken out. Also called “post-tax” deductions.

Agency

Any Department or independent establishment of the Federal Government, including a government-owned or -controlled corporation, that has the authority to hire employees in the competitive, excepted, and senior executive services.

Aggregated

In Enterprise Planning and Simulation, each period the statistical forecast is calculated automatically by the system. A forecast for each individual product can be computed using history for that product. Then these forecasts can be aggregated (that is, summarized) into forecasts for the product family.

Aggregate Reporting

The ability to report time as a collection or mass. In Time and Labor aggregate time reporting features include the ability to report time in a lump sum, as a pattern, in a range of dates, or for an entire crew.

Aging Data

Updating data from separate sources, and separate dates, to a common date using an annualized factor.

Aging ID

A code representing rules for aging open items.

Alias

Any of several PeopleSoft Pension Administration utilities that look up or calculate employee information.

Allocated

In Enterprise Planning and Simulation, the computed forecast and the summarized forecast are two different versions of the statistical forecast. In addition, the forecast at the product family level can be allocated down to the individual products. Usually this allocation is done in proportion to the calculated product forecasts at that level. This version of the (statistical) forecast is called the allocated or prorated statistical forecast.

Allocated Inventory

The inventory assigned to a specific stock request.

Allocation Manager

Perform allocations using the Allocation Manager. Allocations enable you to distribute revenue, expense, and statistical quantities across business units, departments, and so on. You can allocate budget planning to detail levels so that you may perform detailed budgeting. The type of allocation you select determines the output.

Allocation Manager Rules

In the PeopleSoft Enterprise Warehouse, Allocation Manager rules allow you to specify the basis as well as the target tables for moving, aggregating, or multidimensionalizing your output. Rules use Allocation Manager methods to enrich the PeopleSoft Enterprise Warehouse data. *See* Allocation Manager Methods.

Allocation Manager Methods

There are several methods: Arithmetic Operation, Prorata, and Spread Even. Each method enables you to move and/or enrich output.

Allocations

A process of distributing budget amounts to and from other Budget Centers. Budget amounts are allocated to cover, or offset, the costs in one Budget Center by charging them to another Budget Center. An allocation is also the budget amount that is distributed to or from a Budget Center. A budget amount that is charged to another Budget Center appears as a negative amount. This same budget amount appears as a positive amount in the other Budget Center receiving the allocation. PeopleSoft Budgeting-specific.

Allotment

This is a voluntary deduction from pay. Employees may elect up to two allotments from pay, transmitted to a financial institution to the employee's checking or savings account.

ALM (Asset Liability Management)

See PeopleSoft Asset Liability Management.

Allowances

The amount owed to an employee in addition to base salary and which is not defined as part of gross salary. For example, vacation can be considered an allowance. PeopleSoft Budgeting-specific.

Alternate Account

A feature in PeopleSoft General Ledger that enables you to create a statutroy chart of accounts and enter statutory account transactions at the detail transaction level as required for recording and reporting by some national governments.

Alternate BOM

Identifies the multiple ways in which an item can be produced. The primary production BOM is designated as BOM code 1. By using BOM codes, you can associate up to 98 other alternate BOMs with the item.

Alternate Routing

A routing, usually less preferred than the primary routing, but resulting in an identical item. You can specify up to 98 alternate routings for production routing types by entering additional Routing Codes (greater than 1) for the same routing type.

Alternative Minimum Tax (AMT)

AMT is calculated by adjusting the taxpayer's regular taxable income with a number of tax preference items and adjustments. Tax preference items are positive items increasing

Alternative Minimum Taxable Income (AMTI) and are excluded from regular taxable income. Tax preference items include gain from the exercise of incentive stock options.

Amount Type

In PeopleSoft Workforce Analytics, the Amount Type specifies whether a benefits compensation amount is a value or expense, to the employee or the employer.

Analysis Base

Defined static, historical data used both to seed and compare against proposed budgets.

Analysis Group

A grouping of analysis types. Analysis groups can be used for project analysis and grouping or for mapping analysis types.

Analysis Template

A set of pre-defined reports that you can view and publish online. These templates access data in the Enterprise Warehouse tables, and organize it by function, role and industry. The templates allow you to pivot, sort, rank, drill and chart the data, for your analysis needs.

Analysis Type

A 3-character, user-definable identifier that enables you to label the different types of costs. For example, you might want to track budgeted costs (BUD), committed costs (COM), and actual costs (ACT).

Analytical Applications

See PeopleSoft Analytic Applications.

Analytic Forecasting

Analytic Forecasting is the part of the Planning and Simulation feature that creates forecasts for your business requirements.

Annual Amount

In PeopleSoft Workforce Analytics, Annual Amount is an employee-level compensation amount, totaling the calculations for a full fiscal year's worth of accounting periods, in a compensation scenario.

Annual Declaration Report

The French Annual Declaration report is a payroll report which checks establishment profiles to see whether an establishment has to produce the report, and then calculates the amount of all the social security contributions for this establishment.

Annual Leave

Annual leave is absence from work with pay and must be approved by the employee's supervisor in advance. This type of leave (Plan Type 51) is accrued based on years of service: Full-time Permanent/Full-time Seasonal employees ...0-3 years - 4 hours per biweekly pay period; 3-15 years - 6 hours per biweekly pay period (plus an additional 4 hours in the final pay period of the leave year); and 15+ years - 8 hours per biweekly pay period. Part-time Permanent/Part-time Seasonal employees...0-3 years - 1 hour for every 20 hours worked; 3-15 years - 1 hour for every 13 hours worked; 15+ years - 1 hour for every 10 hours worked. Generally, there is a leave year ceiling of 240 hours on accrual; amounts accrued in excess of the ceiling and not used prior to leave year-end are forfeited.

Annual Shareholders Meeting

A meeting of corporation's directors, officers, and shareholders held for the purpose of communicating the operating and financial results for the prior year, the prospects for the future and major decisions of management.

Annual Workforce Survey by Nationality and Professional Category (Enquête sur l'activité et les conditions d'emploi de la main d'oeuvre)

In France, companies are required to submit the Annual Workforce Survey by Nationality and Professional Category to the Ministry of Labor. This report provides an analysis of the company's foreign workforce, which includes any employee who does not have French citizenship.

Annualized Tax Method

A payroll tax calculation method that divides the tax on an annualized amount by the number of pay periods in the year to find withholding for a given pay period, based on the number of withholding allowances. Annualized is the most common tax method.

Annuitant Amount

The gross monthly annuity a federally retired employee receives.

Annuitant CSA Number

A unique number assigned by OPM for a retired employee.

Annuitant Indicator

A code used to indicate the status of an annuitant appointed to a position in the Federal civilian service. Text for the codes is as follows:

1. Reemployed annuitant - Civil Service/FERS
2. Retired military officer receiving pay
3. Retired military non-officer (enlisted) receiving pay
4. Retired military officer receiving pay and a reemployed annuitant - Civil Service

5. Retired military non-officer (enlisted) receiving pay and a reemployed annuitant - Civil Service

6. Not applicable (none of the above)

Annuitant Indicator (cont)

A. Reemployed Annuitant – FERS

B. Former Annuitant - FERS

C. Retired Officer/Reemployed Annuitant - FERS

D. Retired Officer/Former Annuitant - FERS

E. Retired Enlisted/Reemployed Annuitant - FERS

F. Retired Enlisted/Former Annuitant - FERS

Annuity

A series of periodic payments made to an individual. Under a pension plan, these payments are generally made monthly.

Anti-Dilutive

Typically, options or shares where the price is greater than the current fair market value of the security.

APE (Activité Principale Exercée) Codes

APE codes classify the type of industry or activity your French company is in, such as software, banking or insurance. The APE codes are a normalized set of codes that are required by law and are used in regulatory reporting.

API

An Application Programming Interface (API) is the technology that a software product supplies so you can control it or communicate with it from another application. PeopleSoft APIs enable the user to perform desired actions upon PeopleSoft data without having to know the internal logic or rules of the program.

Applicant Hire Process

The procedure of hiring an applicant who has been tracked and administered in the Recruitment pages. Once you assign an Employee ID, the system uses recruitment data to populate the fields in the Personal Data pages.

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.

Application Designer

The integrated development environment used to develop PeopleSoft applications.

Application Engine

PeopleTools batch processes consisting of a set of defined SQL statements. Application Engine processes is more efficient than COBOL or SQR, since they operate within the database system, and don't rely on external processing.

Application Journal Template

A set of rules and default values to control the creation of journals from accounting entries.

Application Processor

The Application Processor is the PeopleTools runtime engine that controls processing of the application from the time the user requests a panel group from an application menu through the time that the database is updated and processing of the panel group is complete.

Application Server

The application server is the centerpiece of PeopleSoft's three-tier architecture. It utilizes Tuxedo, BEA Systems' transaction monitor, to manage client transactions and provide the business rules and workflow capabilities of PeopleSoft's enterprise applications.

Application Server Domain

The collection of server processes and associated resource managers defined by a single PSTUXCFG configuration file. Each application server domain is configured to connect to a single database. Multiple application server domains can exist on the same server machine.

Appointing Authority

The basis that authorized the appointing officer to effect personnel actions on an employee.

Appointing Officer

Denotes if the employee has appointment authority based on laws and regulations.

Approve Time

The Time and Labor feature that approves all employee daily time before it can be sent to payroll for processing. You can approve time by group or by individual employee. You can also unapprove previously approved time.

Approving Official

Individual with the delegated authority responsible for signing the action(s) taken on an employee.

Array

An ordered grouping of data by period and year. PeopleSoft Demand Planning uses arrays in forecasting demand.

Array

Element which enables you to extract information based on a column value. One way of thinking of an array is that it is a SQL statement that retrieves data from an existing table.

Array Dimension

Determines which inventory-stocking possibilities are included in a **Cube View**. This standard one-level dimension consists of the key fields that include, for example, order quantity, safety stock, and turn rate.

Arrears Balance

An amount owed to either the employer or employee, usually the result of a deduction not fully taken.

Ask Price

The price at which someone who owns a security offers to sell it; also known as the asked price.

As-of-Dated

Refers to a snapshot of the data at a given point in time.

Asset Assignment

A streamlined means of associating project costs to assets or asset profiles within PeopleSoft Projects.

Asset Budgeting

Budget for planned asset acquisitions and the associated depreciation expense that can be associated with a Capital Acquisition Plan (CAP).

Asset Catalog

A list of asset profiles which includes information about that asset type, including Cost, Life, Salvage Value, Depreciation Method, Currency Code, and Asset and Depreciation Account.

Asset Category

A standard group of assets. Typical asset categories include Furniture and Fixtures, Machinery and Equipment, Land, Buildings, Leasehold Improvements, and the like. These generally correspond to General Ledger asset accounts. Assets in one category usually share some depreciation characteristics, such as estimated service life and depreciation limits.

Asset Class

An asset group used for reporting purposes. It can be used in conjunction with Category to refine asset classification.

Asset Liability Management

See PeopleSoft Asset Liability Management.

Asset Life

The number of years an asset will depreciate, after which time it might be kept or sold for its Salvage Value. Also *see* Useful Life.

Asset Profile

A template that contains standard depreciation criteria for an asset type and its corresponding asset books. You can use the information in asset profiles as default values when adding assets.

Assignment of Life Insurance

Effective 10/3/94, Federal employees can assign their Basic, Option A and Option B insurance to another person(s), firm(s), or trust(s); Option C is excluded. The assignment of benefits transfers ownership of the FEGLI coverage to the assignee(s). The insured no longer has control over his/her insurance coverage and can no longer designate beneficiaries. Assignment is irrevocable. Either all or none of the insurance can be assigned. Assignment does not have to be to the same person or firm. Assignments must be made in percentages of total insurance versus an assignment of Basic Insurance to one person and Option A to another. Additionally, terminally ill employees can assign their insurance to a Viatical Settlement Firm in exchange for cash (approx. 60% - 85% of the face value of the coverage). Life Expectancy is usually 24 months or less for a Viatical Settlement Agreement.

Assignment Type

This defines the behavior of the object, (resource, activity, or cost object) within PeopleSoft Activity-Based Management. If the object is identified as a source then costs may be allocated from that object to another object, which must be identified as a target. If an object ID is identified as a target it may be allocated costs from another object ID but may not allocate costs. An object ID can be both a source and a target, thereby having the functionality of each.

Associated Primary BOM

With multiple outputs, it's possible that a given co-product can be created in more than one way – in other words, an item is a co-product on more than one items' primary BOM. By assigning an associated primary BOM to a co-product, you are telling the system which BOM to use in exploding the co-product to the next level.

AT Section

In France, this stands for Section Accident du Travail, or Work Accident Section. It is information needed to identify the establishment risk code for insurance purposes.

ATP Reserved Order

An order that has been promised against future supply. The user has an obligation to the customer to fulfill the order quantity by a certain date. ATP-reserved orders are also referred to as *promised orders.*

Attendance

A component of time reporting application whose purpose is to apply business rules related to Benefit Entitlement and Administration and Organizational Administration to time reported as worked or not worked, and to satisfy a variety of reporting needs.

Attendance Reporting

A Time and Labor report that indicates an employee's attendance record. It includes sick leave, vacation time, and other leaves taken.

Attribute

An attribute is an element within a dimension. For example, the element "Store" is an attribute of the dimension "Geography" for the retail industry. An attribute is also a column heading on an analysis and reporting template.

Audit Trail

See Drill-Back Calculation.

Auditor

Person designated to review expense sheets and cash advances before payment.

Automatic Revision Incrementing (Auto Rev)

The ability to automatically set up revision control and generate revisions for revision-controlled items at the business unit level. This includes setting up a revision scheme or a predetermined, ordered list of revision names.

Automatic Spouse Benefit

A joint and survivor pension benefit provided without any actuarial reduction to a pension benefit. The automatic benefit is a n% joint and survivor; the employee is still entitled to choose any optional form of payment and any beneficiary for the remainder of the benefit.

Availability Date

The date a lot becomes acceptable for fulfillment in PeopleSoft Inventory or for consumption in PeopleSoft Production Management. (Availability Date = Creation Date + Availability Lead Time)

Available to Promise (ATP)

The projected supply of a product less the actual demand, which informs the sales and marketing department of the products that can still be sold without modifying the master schedule. ATP isn't cumulative – it's calculated for each period.

Average Daily Balancing

A feature in PeopleSoft General Ledger that enables you to target the ChartFields on which you base average balance calculations, summarize amounts for selected ChartField values according to your reporting requirements, and define the periods for these calculations.

Used by the financial analytic applications in Enterprise Performance Management. For a reporting period (usually monthly) this refers to the average daily balance of an account as opposed to the month-end-balance, which is the balance as of the last day of the month.

Average Daily Balance Ledger (ADB_Ledger)

In the PeopleSoft Enterprise Warehouse, the Average Daily Balance Ledger table (PF_ADB_LEDGER_F00) is similar to the functionality of the PF Ledger table (PF_LEDGER_F00), in that it too supports reporting. However, the Average Daily Balance Ledger is used for average daily balances. It is a table that is used mostly for processes associated with the financial services industry.

Average Inventory

In PeopleSoft Inventory Planning, one half of the average lot size plus the safety stock when demand and lot sizes are expected to be relatively uniform over time. When demand and lot sizes are not uniform, the stock level versus time can be charted to determine the average.

Average Price

The average price derived from either the bid and ask prices (for bid/ask/average) or from the high and low prices (for high/low/average).

Average Static Calc Flag

In PeopleSoft Inventory Planning, a method used with static policies. The average method sets the static policy equal to the weighted-average, time-phased policy over the next argument periods.

Award

A special payment to an employee for certain prescribed kinds of activities or accomplishments.

B

Back Pay Interest

Under certain circumstances, an employee can be eligible to receive additional pay relative to a delayed receipt in salary caused by administrative error in processing a personnel action. The U.S. Office of Personnel Management has established guidelines for Federal agencies on when and how to make these calculations.

Background Process

Any task or process that is grouped with another and runs in the background. Background processes are usually scheduled to run on a regular basis. All background processes are executed through process-specific COBOL programs run outside the Windows environment.

Backlog Reason Code

An identifier indicating the reason an item could not be shipped. Example codes might include out of stock, discontinued, or seasonal.

BAD Forecast Ratio

In PeopleSoft Demand Planning,the maximum acceptable value of the ratio of the and the base component (Standard Deviation/Base Component). When this value is exceeded, the system automatically resets forecast model parameters. The higher the value, the less likely it is that the system will reset the parameters. In most organizations, a BAD ratio of 1.00 or lower is appropriate for most items.

Balance Segmentation

Balance Segmentation is used in Funds Transfer Pricing to divide balances in deposit accounts between core (stable) and non-core (volatile) segments. Core funds represent the minimum balances that are retained on a long-term basis, building a relatively reliable source of funding to the bank. Non-core funds are temporary in nature due to their volatility caused by customer preferences for liquidity, and cannot be utilized on a long-term basis.

Balance Type

Balance Type is a lookup code used to define the type of instrument balances that will be stored in the PeopleSoft Enterprise Warehouse and processed by the analytic applications. Examples of different Balance are Current Balance, Average Daily Balance, Period Ending Balance, or Commitment Balance.

Balanced Scorecard

See PeopleSoft Balanced Scorecard.

BAM

Business Analysis Model. XXX I think this term is incorrect because we use BAM to refer to the application. If we were referring to the business analysis model, we would say BAM model (that is, Business Analysis Modeler model.)

BAM Model

The BAM database published from the template. The model contains both the data and analytic structure used in the application. The BAM database is physically separate from the Enterprise Warehouse database. Data is sent to the model through migration processes.

BAM Template

A file created using BAM design tools, representing the model prior to its creation as a database. This file has an extension of .MDL. This file is published to a BAM database once the model design process is complete. Each application using BAM will deliver templates which the customer will review and publish to a database in their environment.

Bank Identification Number (BIN)

In PeopleSoft Payables, a part of the bank information that identifies business unit banks.

Base Budget

The initial budget defined by the Budget Coordinator. The base budget is distributed as a starting point for Budget to review and edit. The base budget can be zero-based or incremental.

Base Compensation

In PeopleSoft Workforce Analytics, Cash Compensation that is typically categorized as fixed. It includes base pay and shift differentials as well as associated merit, equity, and step increases.

Base Currency

Base Currency is used to consolidate and report financial results of a multinational company. When a company transacts its business operations in different transaction currencies, those currencies are translated to the base currency for reporting purposes.

Base Currency Equivalent (BCE) Amount

If the monetary amount is in a currency other than the base currency, either the Extract-Transform-Load (ETL) process or the Multi Currency Engine can be used to convert the monetary amount to the Base Currency Equivalent (BCE) Amount.

Base Factor

In PeopleSoft Demand Planning, an element of a smoothing constant simulation set that controls base component smoothing in the Model Reset Simulation process.

Base Metric

Metric found on a fact table. A base metric usually contains an aggregate operator, for example “sum” or “count”.

Base Pay

A pay component included in the job comp (job compensation rate) calculation. It is pay for a regularly assigned workweek. For example, you can set up a regular hourly rate plus a shift rate, a union-negotiated rate for hazardous work, and so on.

Base Pay Structure

A PeopleSoft Workforce Rewards module you use to create or revise pay structures, and to assess the cost and impact of implementing new structures.

Base Time Zone

Customer defined time zone used for converting reported time to a common time zone for ease of applying rules (see Time Administration).

Batch

Batch systems are used when realtime updates are not needed. Batch-oriented data collection applications, developed in-house or by a third-party vendor, produce transactions that are collected in an ASCII text file. The text file is fed to a PeopleSoft SQR program that loads the transactions into the database.

Batch Processes

Any of the background programs in the client/server environment of PeopleSoft applications. Batch processes perform operations—such as pay confirmation, deduction calculation, and so forth—on groups of records, and are usually scheduled to run on a regular basis. You run these processes from the Process Scheduler, and they are executed through process-specific COBOL programs.

Before-Tax Deduction

Deduction that reduces net pay and FWT taxable gross, applied prior to the calculation of federal and state/provincial withholding taxes. Also called “pre-tax” deductions.

Begin Calc Date

The date on which PeopleSoft Asset Management begins to deduct from an asset's life.

Begin Depr Date

The date on which PeopleSoft Asset Management begins to calculate depreciation for an asset. Begin Depr Date is calculated using In-Service Date and Prorate convention.

Benchmark Job

In PeopleSoft Workforce Analytics, this refers to a Job Code for which there is corresponding salary survey data from published, third party sources. Jobs for which there is no corresponding salary survey data are referred to as non-benchmark jobs.

Benefit Commencement Date (BCD)

The date on which a pension payee elects to begin receiving payments.

Benefit Deduction

Any amount taken from an employee's pay check to offset all or part of the cost of the employee's benefits.

Benefit Eligibility

The PeopleSoft Pension Administration function that determines if an employee is eligible for retirement or ancillary benefits. A plan may have several retirement types—normal, early, late, death, and disability—each with its own eligibility criteria.

Benefit Entitlement

Any rules governing the circumstances under which employees are entitled to receive certain benefits. Typically, entitlement to benefits is based on type of employee (for example, full time, part time, occasional), length of employment, and specific rules which apply thereto, i. e., work group affiliation, and compensation base. Other criteria may also apply, such as reasons-for-claiming or job performance.

Benefit Formula

The formula that determines a participant's pension benefit in a defined benefit plan, as well as the PeopleSoft Pension Administration function that calculates the benefit.

Benefit Group

Part of a group of defaults assigned to job codes. Benefit group may include medical, dental, and health benefits dependent on individual company parameters.

Benefit Plan

A specific benefit within a plan type. For example, your company's life plan type might include benefit plans of one times salary, two times salary, and three times salary.

Benefit Plan Type

Any category of benefit, such as health, life, or savings.

Benefit Program

A set of benefits and deductions valid for an employee or group of employees. A single company may have any number of programs. An individual employee may belong to only

one program; the deductions and benefits contained in that program are the only valid deductions and benefits for that employee.

Benefit Tables

Any of the tables that contain employee benefits information. These are often relevant to payroll processing.

Benefits Base

The salary used for benefit calculations. The benefits base will be either the employee Annual Rate or Annual Benefits Base Rate.

Benefits Compensation

In PeopleSoft Workforce Analytics, Benefits Compensation is value associated with employment benefits. It can include benefits types for Health and Welfare (Medical, Life Insurance), Retirement (annuities, savings plans, pensions), and Paid Time Off (Vacation Leave, Sick Leave). Benefits compensation is sometimes fixed, and sometimes variable, depending upon the benefit type.

Betriebszählung (Company Statistics Report)

Also called the OFIAMT report. This report provides statistics required by the Swiss Federal Department of Statistics (BFS).

Bias Signal Limit

In PeopleSoft Demand Planning, a number between one and six that indicates how many **Forecast Period** to test for bias. If the bias test is violated, the system records a **Tracking Signals** error in the period up to the number of periods determined by the bias signal limit.

Bias Test

In PeopleSoft Demand Planning, a forecasting test that sets the limit for tripping a **Tracking Signals**. The lower the value, the more likely it is that a tracking signal is set.

Bid Price

The price a prospective buyer is prepared to pay at a particular time for trading a unit of a given security.

BIF file

This is the bulk insert file (input.bif) used with the Verity search engine to specify the documents to be submitted to a collection (search index). It contains a unique key, document size (in bytes), field names and values, and document location in the file system.

Bilan Social Report

See Employee Survey Report.

Bill

In PeopleSoft Billing, any group of bill lines.

Bill Adjustment

The process of making credit or credit and rebill adjustments to an invoiced billing activity.

Bill By Identifier

The Bill By Identifier is used to define how billing activity is grouped when added to a bill through the billing interface or the Populate Billing process.

Bill Header

The record containing information that pertains to the bill as a whole. Each bill has a unique bill header that identifies it within the system.

Bill Inquiry Phone

Bill Inquiry Phone is the number printed on your invoices for your customers to call if they have any questions about their bill.

Bill Line

The basic unit of billing activity representing a billable charge, including the charge identifier, quantity, price, and any other information regarding an individual transaction. Every bill line is related to a bill header that may have one or more bill lines related to it.

Bill Search

A method of finding a bill or bill line when you don't have enough information to call up the bill directly. ***Customer Bill Search*** enables you to locate a bill by Customer Name. You can also choose other parameters to limit your search. With ***Bill Line Search*** you first search for a particular bill and then a line on that bill. Parameters for bill line search include Reference, Date, and Amount.

Bill Source

The point where billing activity originates. Bill sources may be external to the system (imported through the billing interface) or entered directly online. Examples of bill sources include order management, project costing, and contract administration.

Bill To Customer

A customer who receives an invoice.

Bill Type

A category of billing activity variety. Examples of Bill Types include standard and custom order activities.

Bill Update

The process that adjusts bills that have either been entered manually or generated within the system.

Billable Indicator

A status flag that identifies an item as eligible for billing to a customer.

Billback Discount (BB)

A per unit discount which typically requires a customer to perform one or more merchandising activities to receive the discount. A BB discount is not deducted from the customer invoice, but once the customer performs the merchandising activity, a sales representative or broker can approve payment for the discount amount. Billback discounts can originate from a National Allowance or Customer Promotion, and are passed to PeopleSoft Order Management for informational purposes only. Billback discounts are recognized as a liability when the product is shipped.

Billing Location

A number identifying a customer address. Each customer may have multiple locations, but must have one *Primary Location* at which you contact them.

Blackout Period

The period of time, determined by the company, which prohibits certain activity in the company stock. Blackout Periods can affect the trading of some key individuals or can be placed on the entire company.

Bonus Tax Method

Annualizes your year-to-date earnings by multiplying them by the number of pay periods in the year. This method is used for Canadian tax processing.

Book

In PeopleSoft Asset Management, a data location storing financial information—like cost, depreciation attributes, and retirement information—on assets.

Borrow/Loan

The temporary reassignment of an employee to other task reporting or compensation requirements to allow the business to meet unexpected, short-term, fluctuations in staffing or work load. Typically, this kind of reassignment is done informally at a local level, where HR isn't involved and a new job record isn't created. Companies may have specific rules about how long an employee may be borrowed/loaned, how and where productive, non-productive, and compensated absence time will be charged, and what business rules to apply to the borrowed employee's time for the purpose of compensation and benefit entitlement and administration. See also Casual work Assignment.

Bracket

Brackets are a way to look up and retrieve database table values. After you've defined a table, the system finds a corresponding row on that table and returns the value of the bracket. The result is then available for use in other items such as formulas.

Branch

A tree node that rolls up to nodes above it in the hierarchy, as defined in the Tree Manager.

Branch Of Military Service

Identifies, if any, military service in which the employee served.

Breadcrumbs

Breadcrumbs show the navigation path to the current web page location. As you drill down through the different levels of the registry, a "breadcumb trail" appears that shows the path you've selected. Each registry level is separated by an angled brace (>), and you can select any level to navigate directly back to that level.

A typical Breadcrumb would look like this:

```
Home > HR > Administer Workforce > Benefits
```

Break Funding

Charges assessed for mortgages that are paid off before maturity. In the Funds Transfer Pricing (FTP) application, Break Funding charges are factored into the transfer price for a loan that may be prepaid.

Break in Service

A period of time for which an employee does not meet stated service requirements.

Break Price

The price used to determine which options are eligible for repricing. For example, if the break price is $36, then all outstanding option with a grant price of $36 and greater are eligible for repricing.

Break Punch

An in/out punch of when a time reporter takes a break.

Brokers

Individuals or organizations who buy and sell securities. Often they are account executives who work for firms registered with the Stock Exchanges and the SEC. Unlike Transfer Agents, (who are not responsible for sales) Brokers do not maintain records on all your company's certificates. They maintain only sales records and stocks for their clients.

BSC (Balanced Scorecard)

See PeopleSoft Balanced Scorecard.

Budget Activity

A type of activity peformed using PeopleSoft Budget Planning. Budget activities include Line Item Budgeting, Line Item Mass Adjustments, Budget Allocations, and Position Budgeting. PeopleSoft Budget Planing-specific.

Budget Amount Ledger

Stores budget amounts and is updated by posting budget entries, transfers, and adjustments.

Budget Analyst

A role within PeopleSoft Budgeting. Budget Analysts are typically people within an organization responsible for reviewing and analyzing a prepared budget before submitting it to the Budget Coordinator. PeopleSoft Budgeting-specific.

Budgetary Account Only

An account used by the system only and not by users; this type of account will not accept transactions. You can only budget with this account. Formerly called "system-maintained account."

Budget Category

A set of related expenses that are accumulated for proposal budgets and reporting to a sponsor. The estimated cost for a set or class of accounts.

Budget Category

Numeric/alpha identification given to each category of positions.

Budget Center

In PeopleSoft Budgets, any entity responsible for producing or reviewing budget data. For example, a Budget Center might be the individual departments responsible for producing budgets.

Budget Center Dimension

In PeopleSoft Budgets, the dimension by which you distribute budget data. If you budget by department, your department dimension will be your Budget Center Dimension. You'll assign Budgets Users to the nodes and detail values on the tree you use to build your Budgets Center Dimension.

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 Check Override

Selective suspension of Budget Processing. With this feature you can override the controlled budget for a transaction that failed budget checking due to insufficient funds; or override the tolerance limits for a transaction rejected due to exceeded tolerance limits. When you push the Override button, the system flags the transaction to allow the Budget Processor to process successfully regardless of available funding. You can cancel the override any time before the Budget Processor is run by clicking the Cancel Override button.

Budget Control

In commitment control, it ensures that commitments and expenditures don't exceed budgets**.** It enables you to track transactions against corresponding budgets and abort 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 Coordinator

A role within PeopleSoft Budgeting. Budget coordinators are responsible for monitoring the budget process. The Budget Coordinator is typically located within an organization's central budget office and builds the budgeting model. PeopleSoft Budgeting-specific.

Budget Detail

A level of itemization that when combined makes up a major budget category.

Budgeted Rates

In PeopleSoft Activity-Based Management, the rate your organization uses based on the budget.

Budget Error Exception

A transaction that fails budget checking, causing an Error or Warning to be issued. See **Error Exception** and **Warning Exception**.

Budgeting Functions

PeopleSoft Budgeting's six main action categories, including: system administration, budgeting setup, budgeting preparation, budgeting analysis, data integration and my profile. Your user role determines how many of these functions display and are available.

Budgeting Model

The framework for an organization's budget development process. Business unit defines a Budgeting Model. The Budget Coordinator typically defines the model and includes the time period of a budget cycle, time period for phases within a budget cycle, the sources of data that will be available to budget users, the methods that will apply to line-item budgets, and other budget options and control parameters. PeopleSoft Budgeting-specific.

Budgeting Type

Associated with the budget ledger type set definition, a budget type is an indication of whether the organization uses a standard budget ledger, project budget ledger, or controlled budget ledger for budgeting.

Budget Justification

Written explanation further defining the what and why of a budget category.

Budget Period

The period in which you define plans to meet your organizations training requirements.

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.

Budget Phase

In PeopleSoft Budgets, a span of time during which a budget or portion of a budget is to be completed. You'll filter dimensions, assign alternate Budgets Users, enable Position and Asset budgeting, and specify Budgets User notification options at the Phase level.

Budget Plan

In PeopleSoft Workforce Rewards, when working with a Compensation Planning BAM model. A budget plan is a rollup of like compensation rules. For example, for base pay rules budget plans are a rollup of values for like Action Reasons. For variable pay rules budget plans are a rollup of the values for like Variable Compensation Plan IDs.

Budget Preparer

A role within PeopleSoft Budgeting. Budget preparers are typically people within an organization responsible for developing the detailed budget for a Budget Center and submitting it to a Budget Reviewer or Analyst for review and approval. PeopleSoft Budgeting-specific.

Budget Reviewer

A role within PeopleSoft Budgeting. Budget reviewers are typically people within an organization responsible for reviewing and approving a prepared budget submitted by a Budget Preparer. PeopleSoft Budgeting-specific.

Budget Seeding

Represents a new budget or forecast, such as historical data that is manipulated to develop a more current representation for a proposed budget. Uses detail data as the budget seed or basis to create the base budget that represent the level of detail in which budget numbers are prepared.

Budget Translation Trees

Trees translate (summarize) source transactions into the appropriate levels for processing against control budgets. This is because you usually budget above the level of your source transaction ChartFields on a tree.

Budget Type

Indicates whether a budget is for expenditures or revenues.

Budget Warning

See **Warning Exception**.

Budgets User

In PeopleSoft Budgets, any user who needs to gain access to the Budgets. You'll designate Budgets Users on the Budgets Users page through the Coordinate Budgets window. You'll also assign these users to the tree representing your Budget Center Dimension.

Budget View

A user-defined view where selected dimensions, columns and rows of data determine the layout of line-item budgets affecting the view or entry of data.

Budget Year

The institutionally defined, consecutive, 12-month period to which a financial transaction or summary applies.

Build Option

A detailed PeopleSoft Planning model that specifies a method of building an assembly item. This model specifies the routing, resources, and materials that are necessary to produce the item.

Built-in function

Prior to PeopleTools 8.0, there were only built-in functions, like FetchValue, ScrollSelect, etc. A built-in function, in your code, is on a line by itself, and doesn't (generally) have any dependencies. You don't have to instantiate anything before you can use a built-in.

Business Interlink Definition

A definition encapsulating an external Transaction or Query and providing a set of generically typed input/outputs that can be assigned to PeopleCode variable or Record Fields at runtime. A Business Interlink Definition is added to the Application Designer's objects at the same level as Fields, Records, Panels, etc.

Business Interlink Design-Time Plug-in

An XML file that, when coded for an external system, encapsulate that external system and provide a catalog of Transactions, Classes and Criteria specific and meaningful to that external system.

Business Interlink Framework

The framework for integrating any external system with PeopleTools application objects. It is composed of the following components:

1) An External System, 2) Generic definitions for a Transaction/Query command interfaces, 4) Business Interlink Definitions, 4) Business Interlink Plug-in.

Business Interlink Object

An instantiation based on a Business Interlink Definition. Actual data can be added to the inputs of the Business Interlink Objects once the appropriate bindings are provided. The Business Interlink Object can be executed to perform the external service. Once a Business Interlink Object is executed, the user of that object can retrieve the outputs of the external service. The Business Interlink Objects use buffers to receive input and send output. When a Business Interlink Object is executed, the transaction/query/class associated to the Business Interlink Object will be executed once per each row of the input buffers corresponding to the input Records. If there is only one row, after appropriate substitution by the driver, it is executed only once.

Business Interlink Runtime Plug-in

A set of C++, Visual Basic, or other high-level language methods that, when coded for an external system, encapsulate that external system and provide the execution methods to match the Business Interlink Design-Time Plug-in. (The catalog of Transactions, Classes and Criteria provided by the Design-Time Plug-in can also be provided by the Runtime Plug-in.)

Business Objects

A way of identifying those mass changes that have been designed to be referenced by a flexible formula and provide them with a shorter name to simplify the creation of flexible formulas.

Business Planning

The type of planning that focuses on elimination activities that are not needed by changing the drivers.

Business Rules

Rules that can process information differently depending on the values of data in the PeopleSoft Enterprise Warehouse.

Business Unit

A corporation or a subset of a corporation that is independent with regard to one or more operational or accounting functions. PeopleSoft General Ledger business units typically comprise individual entities for accounting purposes.

Business units in PeopleSoft Projects represent operational structures but not necessarily independent financial units.

PeopleSoft Payables business units are either *Vouching* (have payables accrued to them) or *Charge to* (have voucher expense distributions charged to them), and pass journals to general ledger units.

PeopleSoft Purchasing business units share vendor, purchase order, and receiving information with PeopleSoft Payables units in the same SetID.

A PeopleSoft Inventory business unit is a storage facility that maintains its own replenishment and costing methods, as well as its own definitions and guidelines.

The Manufacturing business unit must be identical to the Inventory business unit in order to link the manufacturing and inventory processes.

The Order Management business unit controls certain order processing parameters (tax and freight calculation methods, base currency, credit card hold options, and so on) for its associated PeopleSoft eStore and Mobile Order Management merchant variants.

Business Unit Audit List

One or more business units specifically targeted for expense report and cash advance audits.

Buying Agreement

You can structure flexible and easy-to-use buying agreements for customers or groups of customers. You can set up maximum amounts and specify the minimum dollar value per order placed against it. You can automatically generate sales orders or create sales orders online from buying agreements. Rebate and penalty calculations can be implemented for buying agreements.

C

Cafeteria-Style Benefits

Any programs offering several benefit plans from which participants make elections. Cafeteria-style benefits may or may not include flexible credits.

Calculation

In PeopleSoft Pension Administration, the determination of a participant's pension benefit.

Calculation Rule

Criteria for calculating benefits, including as-of dates for age, service, premium, and coverage calculations; rounding rules; and minimum and maximum coverage amounts. Any number of program and plan combinations can use a single set of calculation rules.

Calculation Rule [Global Payroll]

Any rule you develop using combinations of elements to command the system to perform a type of calculation.

Calendar

In PeopleSoft Manufacturing, a list defining the days your enterprise is available and the hours of operation for each day. The system first looks to see whether you are using a work center specific calendar. If none is defined, it looks at the production calendar. If no production calendar is defined, planning and scheduling functions base start and due dates on a five-day workweek.

In PeopleSoft Demand Planning and Inventory Planning, a list defining the start and end dates for each time-phased period. It also contains daily weights for distributing raw data into different period buckets.

In PeopleSoft General Ledger, your accounting calendar defines the time periods to which you post transactions for different ledger group and business unit combinations. You can have multiple calendars, so you can keep a calendar for actuals, another for budget and forecast activity, and still others for special reporting or transitional needs.

Calendar Group ID

Allows you to group together multiple Calendars that you want to run together at the same time. It also controls the order in which the Calendars are processed. You can only group calendars together that are for the same country (based on pay entity country).

Calendar Scope

A time period type (Day-Factored, Month-Factored, or Week-Factored) for use in building your time period calendar.

Canada Academic Teaching Surveys

Statistics Canada requires that all Canadian universities (all degree granting institutions) produce full-time and part-time *Canada Academic Teaching Surveys*. These reports are a legislative requirement. PeopleSoft HRMS 8 provides you with the functionality to code HRMS information using Statistics Canada codes and create both the full-time and part-time Academic Teaching Surveys.

Canadian Industrial Sector

The Canadian industrial classification code with which employees are associated for Canadian employment equity reporting purposes.

Canadian National Occupational Classification (NOC) Codes

NOC codes are occupational classification codes for Canadian companies provided by the government.

Canadian Standard Occupational Classification (SOC) Codes

SOC codes are occupational classification codes for Canadian companies provided by the government.

Cancellation

A process that terminates stock fulfillment requests, allowing reserved and allocated items to be returned to inventory.

Cancellation

In the context of an employee stock plan, a transaction (usually triggered by a specific event, such as a termination of employment) in which outstanding securities are declared void and inactive and returned to the pool of securities reserved for issuance under the plan or retired.

Candidate Keys

In PeopleSoft Demand Planning, elements of data that can be used to construct the **Forecast Item** key field at different levels of the forecast.

Capacity Rate

A rate you assign to a capacity cost object. This enables you to track and report on excess capacity.

Capacity Fence

A time fence that indicates that date and time after which PeopleSoft Enterprise Planning or Production Planning solvers ignore capacity violations. The solvers do not use this date in processing capacity violations.

Capacity Multipliers

A multiple used in PeopleSoft Enterprise Planning and Production Planning to determine the available capacity on a resource. Since a capacity multiplier is effective-dated, you can use it to vary the resource's available capacity over time.

Capital Acquisition Plan (CAP)

A method of projecting and tracking capital expenditures for a project. Budgeted assets and actual expenditures can be associated with a CAP Plan so the owner can track planned against actual costs.

Capital Gain

The difference between an asset's purchase price and selling price, when the difference is positive. Capital gains can be either short-term (where the capital asset was held for 12 months or less) or long-term (where the capital asset was held for 12 months or more).

Capital Gains Tax

A tax on profits from appreciation in owned real property, recognized at the time the property is sold; real property includes owned company shares.

Capitalization

The total types and amount of the outstanding securities that have been issued by a corporation. Generally includes both equity and debt securities.

Capital Markets Instrument

In the financial services industry, Capital Market Instruments are assorted financial instruments issued by organizations to raise capital for funding operations. Participants are made up of interested parties that choose to supply or acquire the capital funding through such vehicles. Derivatives, debt instruments, equities and foreign exchange instruments that are traded in highly liquid markets represent the instruments. In the PeopleSoft financial analytic applications, Capital Market securities refer to instruments that are bought/sold by the institution for its own investment account. The capital markets set the product prices and interest rates.

CAP Sequence Number

The number that distinguishes a small project belonging to a CAP plan. Budgeted assets can be associated with an overall CAP Plan and a CAP Sequence, if that level of detailed tracking is desired.

Carry-Forward

Residual contributions that remain in a stock purchase participant's account after the purchase of shares that are used toward future purchases.

Carrying Cost

In PeopleSoft Inventory Planning, a value that shows the cost associated with holding a dollar of inventory for one year. The value is presented as a percentage.

Case Officer

In Germany employees in your company are designated as Case Officers, and have responsibilities for handling health and safety incidents.

Cash Balance Accounts

The PeopleSoft Pension Administration function that tracks the activity in an employee's hypothetical account under a cash balance plan.

Cash Balance Plan

A defined benefit plan designed to look like a defined contributory plan. The plan periodically credits a percentage of pay to each employee's hypothetical account.

Cash Compensation

In PeopleSoft Workforce Analytics, Cash Compensation is a component of direct compensation. Cash Compensation consists of direct cash payments made to an employee for base compensation and short-term variable compensation.

Cash Exercise

At the time of exercise, the optionee is required to pay in cash the total option price plus any withholding taxes due to the company.

Cash Flow Generator

This is a support module for the PeopleSoft financial services analytic applications. It generates actual and projected cash flows for financial instruments by using output from the other support modules, such as loan prepayment rates, deposit runoff rates, product pricing indices, discount rates, and product definitions (such as start and end dates, balance amount, interest rate, term, payment dates, repricing and compounding frequency, and accrual basis) to generate the cash flows. The Financial Performance Measures module accesses the cash flow results to calculate the required financial measures.

Casual Preparer

An additional user role at the lowest level of budget preparation for a budget center. This user performs the same activities as the Budget Preparer role when access is granted. The system does not, however, enable the Casual Preparer role to define their own private views for line-item budgeting.

Casual Work Assignment

The temporary assignment of an employee to a work position or location to meet the needs of the business. Typically, there is no Human Resource activity to support the work assignment (that is, a new Job record is NOT created). Often compensation rules that accrue to the temporary assignment override the compensation rules that apply to the employee's normal work assignment. See also Borrow/Loan.

Catalog

The list of transactions, classes, and queries used to interface to the external system. Integration users are presented with this list when they pick the type of Business Interlink Plug-in they are going to use. There are four types of catalogs: transaction, class, operator, and configuration parameter.

Catalog

A way of organizing your training courses into classifications for increased flexibility. Catalogs consist of categories and subcategories.

Category

Categories are the primary level of a two-tier structure of training courses. Categories can consist of subcategories that provide further course definition.

Category Tree

A hierarchical structure that groups products by category to control how they are displayed in PeopleSoft eStore web pages. Used also by Mobile Order Management to enable product information to be accessed by a wireless device.

CBM

See PeopleSoft Customer Behavior Modeling.

Census Metropolitan Area (CMA) Code

In Canada this code is prescribed by the government and refers to the area of an urbanized core with a population of at least 100,000.

Central Personnel Data File (CPDF)

Two types of reporting made by agencies to the OPM include the Dynamic and Status files (quarterly and monthly, respectively) covering a range of employee personnel/payroll data.

Certain and Continuous Payment Option

A form of pension payment where the benefit is paid out for the lifetime of the participant with a specified number of payments guaranteed so that a beneficiary will receive payments until the end of the guarantee period if the employee dies before the guaranteed payments are complete. For example, under a ten-year certain and continuous payment option, a retiree who lives less than ten years receives payments until death, then the retiree's beneficiary continues to receive payments for the remainder of the ten year period. A retiree who lives longer than ten years continues receiving payments after the ten year period until death. Also knows as a "Term Certain and Continuous" payment option.

Certain Only Payment Option

A form of pension payment where the benefit is paid out entirely over a specified period of time—usually five, ten, or fifteen years—with no ongoing payments after the specified period. If the retiree dies before payment period is over, the remaining payments are made to a beneficiary. Also known as a "Term Certain" payment option.

Change To Lower Grade

- For positions under the General Schedule or under the same wage grade schedule, a change-to-lower grade changes the employee to a lower grade; and
- When both the old and new positions are under the same type ungraded wage schedule, or in different pay-method categories, a change-to-lower grade changes the employee to a position with a lower rate of basic pay.

Charge Out

A **Material Issue** used when the item is scheduled for future return.

ChartField

A field storing 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

PeopleSoft enables you to set up ChartFields and indicate that you want specific ChartFields to match (balance) on the debit and the credit side of a transaction. When you work with Controlled Budgets, the Fund and Budget Period are already set up in the system to balance (match). For example, suppose you want to balance by Class and Program. You indicate that these on a panel that these ChartFields are required, along with Fund and Budget Period which should already be selected. When you enter a transaction, you must enter the same Class, Program, Fund, and Budget Period ChartFields on both sides of the accounting entry. but you can modify any ChartFields, other than these four, on the user-defined line. The system always requires that total debits equal credits.

ChartField Combination Edit

Also called *Combo 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.

ChartViews

Charts of data in the model, presented through the Worksheet which retains the ability to drag dimensions on the chart as desired.

Check In/ Check Out

The process of retrieving planning activities from the BAM database (check out) and posting changes and results back into the database (check in).

Child

A node or detail on a tree linked to another, higher-level node (referred to as the parent). Child nodes—projects, customers, and so on—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.

Child

A node or detail of a tree linked to another, higher-level node referred to as the parent. Child nodes can be rolled up into their parent. A node can be a child and a parent at the same time depending on its location within the tree.

Chunking

Chunking is a PeopleSoft Enterprise Warehouse mechanism that makes voluminous processing easier through the use of multiple small parallel processes. By enabling chunking, multiple jobs are spawned from one Jobstream. These jobs run in parallel (behind the scenes) to process data efficiently.

Citizenship Code

Numeric indicator as to whether the employee is a U.S. citizen or a foreign national serving in the U.S. The codes are:

- citizen
- other

Civil Service Retirement System (CSRS)

A retirement plan available to employees of the federal government. CSRS covers all employees appointed to a position in the federal government before January 1, 1984. Coverage includes a basic annuity plan with employee contributions and the Medicare Hospital Insurance component (1.45%) of the Social Security tax.

Class catalog

Lists classes used to interface to an external system. A class contains data members of basic types and/or objects that are typed after another class. A Class can also contain lists of basic types or objects.

Class ChartField

A ChartField value that identifies a unique appropriation budget key when you combine it with a Fund, DeptID, and Program Code as well as a Budget Period. Formerly called “sub-classification.

Classification Code

Need App A code that categorizes an engineering change. Example classification codes include the following: Mandatory, Optional, Upgrade, Quality, and Safety.

Clock Hour Reporting

Method of reporting time by recording actual times in and out (start and stop) (see Time Reporting).

Clone

To create a unique copy of an object. When used in PeopleCode, clone will always mean to make a unique copy. Copy, on the other hand, may or may not mean making a unique copy. Copy may mean making a new reference to an object, so if the underlying object is changed, both the copy and the original change.

Cloning

The process that enables you to copy run controls to create employee schedules from existing Run Control ID's that have already been executed and saved.

Close Date

The date in which time entry is no longer allowed for a given pay period. Defined as an offset number of days to the pay period end date.

Close Price

The price of the final trade for a security at the end of the trading day.

Closure Calendar

A calendar that establishes closure dates for shipping, receiving, and materials management operations for a specific **Business Unit**. Typically, application processes account for these closure dates when determining Lead Time and dates for anticipated fulfillment processing dates (scheduled shipment dates, scheduled arrival dates, and lot retest dates, for example).

CMA (Census Metropolitan Area) Code

In PeopleSoft Workforce Analytics, the CMA code is prescribed by Statistics Canada, and refers to the main labor market area of an urbanized core with a population of at least 100,000.

COBRA (Consolidated Omnibus Budget Reconciliation Act)

In PeopleSoft Workforce Analytics, this refers to legislation that requires employers to offer continued health care coverage to employees, and their dependents, who lose benefits coverage under certain defined conditions such as voluntary termination, divorce, becoming an overage dependent, or retirement. Any individual, whether employee or dependent, that is covered under a health plan at the time of a qualifying event, has the option to elect COBRA coverage.

Codepage

One character set.

Collection

To make a set of documents available for searching in Verity, you must first create one or more collections. 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 matching various search criteria. A collection is a set of statistics and pointers to the source documents, stored

in a proprietary format on a file server. Since a collection can only store information for a single locale, PeopleSoft maintains a set of collocations (one per language code) for each search index object.

Combined Federal Campaign (CFC)

A vehicle used by federal employees to contribute to a charity or charities of their choice.

Commercial-Off-The-Shelf (COTS)

Equipment or software that is currently sold commercially to at least one customer.

Commission Tax Method

A payroll tax calculation method that adds year-to-date earnings to earnings for this pay period and finds the annualized gross by multiplying by the number of pay periods in the year; the gross is then divided by the number of tax periods specified on the paysheet. This method is used for Canadian processing only.

Commitment Control

Commitment control includes budget control and commitment accounting functionality.

Common Shares Issued and Outstanding

Represents the residual ownership interests in the corporation. This is the composite number of shares available and tradable on the open market.

Community Background

In the United Kingdom Community Background refers to the religious category, such as Catholic or Protestant, of employees, job applicants or appointees. See the Northern Ireland Report for more information.

Compa-Ratio

In PeopleSoft Workforce Analytics, Compa-Ratio is most commonly defined as the relationship between current pay and the midpoint calculated as: (Incumbent Pay/Midpoint)*100. Usually expressed in whole numbers, or in percentage form by dropping the multiplication operation. Much less common is the use of a compa-ratio calculation as: range midpoint/market rate.

Compensation Frequency

In PeopleSoft Workforce Analytics, this is the frequency at which a job is paid. This is the value you use for reporting or quoting pay. Examples include Annually, Monthly and Weekly.

Compensation Planning

In PeopleSoft Workforce Analytics, this is the process through which employee compensation plans are defined, and compensation budgets are allocated throughout an organization. Major components of compensation planning include designing pay structures, setting individual pay levels, and budgeting and forecasting compensation spending.

Compensation Rate

In PeopleSoft Workforce Analytics, this is the compensation rate for a job. This is the rate the company uses for quoting and reporting pay.

Comp time (compensatory time)

A PeopleSoft Time and Labor-managed employee benefit where time off is granted in exchange for time worked based on customer-defined criteria; is associated with an expiration and is used as reported time (see Attendance).

Compensation

The process by which a worker is remunerated for services rendered to, or work performed on behalf of a business entity.

Compensation Package

All of the base and non-base components on a job row.

Compensation Rules

Business methodology or logical process that is applied to reported time in order to determine payable time (see Time Administration).Competency

In PeopleSoft Workforce Analytics, Competency is a knowledge, ability, skill, accomplishment, or National Vocational Qualification (NVQ).

Competency Inventory

All of the roles, tasks, competencies and accomplishments possessed by the workers in the current workforce. This data is migrated from internal source systems into the data warehouse tables of the PeopleSoft Enterprise Warehouse.

Competency Strategy

The type and number of roles, tasks, competencies and accomplishments essential to accomplishing a business scenario based on your strategic business goals.

Competitive Appointment

An appointment to a position in the competitive service following open competitive examination or under direct-hire authority. The competitive examination, that is open to all applicants, may consist of a written test, an evaluation of an applicant's education and

experience, and/or an evaluation of other attributes necessary for successful performance in the position to be filled.

Competitive Service

All positions as defined by 5 USC 2102 in the executive branch of the Federal Government are in the competitive service unless they are specifically excluded from it. Positions in the legislative and judicial branches are outside of the competitive service unless they are specifically included.

Compress

The act of placing a Planning task as early as possible in the schedule without violating any constraints.

Compressed Split

In PeopleSoft Demand Planning, an optional function that allows a split database to be compressed so it can be transferred to an account manager's computer.

Concurrent Offerings

Multiple stock purchase offerings that are active and outstanding at the same time. The end date is measured from the employee's grant date.

Concurrent Processing

The situation in which you run multiple batch processes at a time. In PeopleSoft Benefits Administration, for example, simultaneous open enrollment and event maintenance qualifies as concurrent processing.

Configuration Code

A unique 50-character identification code that accurately tracks and costs inventory with the PeopleSoft Product Configurator. It corresponds to a lot number for a non-configured item.

Configuration Costing

The overall process of reviewing and evaluating anticipated cost data for a configured item.

Configuration parameter catalog

Used to configure an external system with PeopleSoft. For example, it might set up configuration and communication parameters for an external server.

Consolidate Assets

In PeopleSoft Asset Management, the process of consolidating multiple load lines, usually coming from a separate application, into one asset.

Consolidate Depreciation

In PeopleSoft Asset Management, the process of summing all open Add and Adj transactions by transaction type, **Transaction Date**, and accounting date for all composite members reporting to one composite asset.

Consolidated Bill

A grouping of bills gathered together for invoice presentation. The bills belonging to a consolidated bill are invoiced and printed together, with a page summarizing the bills as a group.

Consolidations

The PeopleSoft Pension Administration functions that accumulate hours, earnings, and pension contributions based on payroll data.

Consolidations-Elimination Set

A related group of intercompany accounts that is processed during consolidations. Once eliminated, this group of accounts should normally net to zero.

Constraint

A limit to a schedule, that, when violated, must be repaired to produce a valid schedule. User-configurable Planning constraints include Missed Request Dates, Missed Promise Dates, BI Shortages, RM Shortages, Capacity Overloads, Missed Inventory Targets, Changeovers, and Excess Inventory. See also **Temporal Constraint**.

Constraints

In the PeopleSoft Enterprise Warehouse, a constraint can consist of one or more filters and is used to define complex business logic. Constraints are based on DataMaps.

Consumption Pattern

In PeopleSoft Activity-Based Management, an attribute used to describe how an activity interacts with the cost objects to which it has been assigned. A unit type activity can expect to be performed on a regular basis so that each time a product is produced. A batch type activity may only be performed periodically for a given range of transactions. For example, each time a machine is setup to produce another product type. Sustaining type activities generally occur to support the overall operation of a company unrelated to products produced or customers served.

Contact

A person associated with a Customer ID. Contacts can be internal contacts or external contacts. Internal contacts are your employees who manage the relationship with your customers, from handling billing inquiries to product/warranty questions, to basic product/service questions. Interactions with customers can be recorded via PeopleSoft Conversations. Self service interactions can be recorded through PeopleSoft Contact Us. External contacts are your customer's representatives who can access self-service transactions

and receive documents such as sales order acknowledgements. Contacts must have a User ID to access self-service transactions.

Contact Us

A method by which customers and unregistered guest users send email messages to specific addresses or members of the merchant's organization. Merchants can also define automatic response messages.

Container

An Inventory stock unit for receiving, putaway, bin to bin transfers, picking, shipping, adjustments, and physical accounting. Each container is associated with a unique container ID.

Content Reference

Content references are pointers to some kind of content registered in the portal registry. These are typically either URLs or iScripts. Content references fall into three broad categories: target content, templates, and template pagelets.

Contextual reference

PeopleCode refers to a row or buffer field determined by the current context; that is, the context in which the PeopleCode program is currently executing.

Contingent Beneficiary

In PeopleSoft Pension Administration, any non-spouse pension beneficiary, including a child, other relative, or a trust. Spousal consent is required in order for an employee to name a contingent beneficiary.

Contracting Officer (CO)

Individual who has the authority and the official responsibility to produce a sound acquisition document.

Contracting Officer's Technical Representative (COTR)

Individual responsible for monitoring a contract and its associated tasks and deliverables.

Contractor

Any individual or non-employee reporting time that will not be paid through the payroll system.

Contribution

Represents money a stock purchase participant elects to contribute to the plan. Contributions are deducted from the participant's paycheck and used to purchase stock pursuant to the offering and purchase period they are enrolled.

Contributory Plan

A Pension plan to which employees contribute. Contributions are typically a percentage of pay deducted from the employees' paychecks.

Control Budget

Commitment control enables you to establish budgets that provide extensive, active budgetary controls over transactions, rather than just passively recording transactions.

Control ChartField

A control ChartField is a key ChartField that you designate to be the control field. Designating a ChartField as the control allows you to set attributes for a specific value of the ChartField that are different from the attributes specified for the budget type in general. For example, if the tolerance for a Projects budget type is set to 10% in general, you can override this value, making it higher or lower for specific projects.

Control Group

A mechanism to relate vouchers together for the purpose of controlling voucher input into PeopleSoft Payables. Generally used for assigning vouchers to data entry personnel and for reviewing input.

A set of parameters that determines the major forecast process options. The Control Group code is assigned to a group of **Forecast Item** and controls the forecast development and tracking for each item in the group.

Control groups are used by the Analytic Forecast Component to govern particular properties of the forecast rule, such as what accuracy to expect and what statistical method to apply. Forecast elements are assigned to exactly one control group. They manage differences among forecasts within a set.

Control Hierarchy

The relationship between business units, origins, vendors, and control groups in PeopleSoft Payables that defines which processing data will be automatically entered on each voucher.

Control Number

A sequential identifying number used to identify an exercise.

Control Plan

In PeopleSoft Quality, a plan that brings together application, measurement, and control and response criteria for a specific product and process.

Conversation

Any notes, transcript, or detail of a telephone call between an employee and a customer. Conversations may be tied to items, payments, purchase orders, document references, or bills of lading.

Conversion data profile

A conversion data profile takes the values from a particular PeopleSoft database table (such as the table holding bank transaction codes) and specifies how that value appears in PeopleSoft Business Documents.

Conversion data profile

A conversion data profile takes the values from a particular PeopleSoft database table (such as the table holding bank transaction codes) and specifies how that value appears in PeopleSoft Business Documents.

Conversion Loader

A sample SQR delivered with PeopleSoft Asset Management that transfers data from multiple fixed-length ASCII files into sample, relational conversion tables.

Copy Bill

In PeopleSoft Billing, the online environment providing for the replication of a single bill, generating a new bill with its own unique invoice number.

Core Functionality

Core functionality is the set of information in PeopleSoft HRMS that is common to your entire global workforce tracking needs—and is always displayed on the primary page.

Core hours

The hours a workday, workweek or pay period in which a time reporter must be present for work in a flexible work schedule (see Scheduling).

Corporate Account

In PeopleSoft applications, this is equivalent to the Account (ACCOUNT) ChartField. The term is used to make a distinction 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.

Corporate Reporting

Companies with more than $10 million in assets whose securities are held by more than 500 owners must file annual and other periodic reports. Publicly held companies are required to file documents with the SEC which include:

- Registration statements for newly-offered securities
- Annual and quarterly filings (Forms 10-K and 10-Q)
- Proxy materials sent to shareholders before an annual meeting

- Annual reports to shareholders
- Documents concerning tender offers (a tender offer is an offer to buy a large number of shares of a corporation, usually at a premium above the current market price)
- Filings related to mergers and acquisitions

Corporate Repurchase

When a corporation elects to repurchase some of it's own securities. This reduces the Common Shares Issued and Outstanding. Typically, used to improve the valuation of the company's common securities outstanding as well as the Earnings Per Share (EPS).

Correction to IRR

An IRR type used when corrections need to be made to an original IRR that has already been submitted to the Office of Personnel Management (OPM). Federal employees covered by the CSRS retirement plan require SF-2806-1. Federal employees covered by the FERS retirement plan require SF-3101. A Correction IRR is also used if original retirement deductions were over-reported. See also Individual Retirement Record (IRR).

Correspondence Customer

A customer to whom all correspondence (statements) is addressed, often a corporate customer receiving correspondence for associated child customers.

Cost Accounting

A method where business costs are accumulated and distributed to products, processes, or discrete undertakings on an equitable basis. There are a variety of cost accounting methods, but they all share the same basic functions. classifying costs, recording costs, allocating costs to products or activities, summarizing and reporting costs to management. Cost accounting requirements and financial accounting requirements are not necessarily synonymous.

Cost Assignment

Resources assigned to cost objects or activities.

Cost Basis

Typically, this refers to the original price of an asset used in determining capital gains. However, in the case of death of an optionee, the appraised value of the asset at the time of death is the cost basis.

Cost Center

A Time and Labor Business Unit, in which all related costs attributable to some center within a business (such as an activity, an organization, or a program), are segregated for accounting or reimbursement purposes.

Cost Element

See **Inventory Cost Element** and **Manufacturing Cost Element**.

Cost Flow

Determines how depletions will occur for purposes of costing a transaction. Cost flows available include Specific Lot ID, Specific Serial ID, FIFO, and LIFO.

Cost Objects

Cost objects represent cost information about products, customers, and channels. They are the final results of the activities performed by your business, representing the focal point of costing and profitability analysis. Examples are products, customers and channels. They are the final results of the activities performed by your business. Your model's resources and activities are linked to the cost objects. They are often the focal point of profitability analysis.

Cost of Capital

An attribute used to describe the behavior of a particular cost object. A primary cost object is typically the main focus of the activity-based management analysis. This may be a product, customer or channel that you wish to calculate cost for. A support cost object may be used in a similar manner but may be further allocated to other support cost objects or primary cost objects.

Cost Of Living Allowance (COLA), Non-Foreign

A cost-of-living allowance payable to an employee at a location in a non-foreign area where living costs are substantially higher than those in the Washington, DC area.

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 Profile Group

A grouping of items for the purpose of costing transactions and valuing inventory for a given book. Assigning an item to a cost profile group determines the books used by the item when accounting for that item.

Cost Roll-up

A process for calculating item costs. Cost roll-up provides a summation of all of the costs associated with the bill of material structure and the routing used in producing the item.

Cost Row

A cost transaction and amount for a set of ChartFields.

Cost Template

A collection of cost components that you can apply to a group of purchased items.

Cost Type

A user-defined method of categorizing item costs in Manufacturing for simulations and what-if analysis. Examples of cost types include current costs (which reflect the item's current bill of material or routing), proposed costs (which could be used in preparation for the next standard cost period), or activity-based costs (which include costs for items that consume a given activity).

Cost Version Type

A combination of cost types and cost versions used in cost rollups. Valid values include production (rolls up only manufacturing data and uses only the primary BOM and routing, each with a code of 1), engineering (can roll up with either manufacturing or engineering data, with any combination of BOM/routing codes), or simulation (only rolls up with manufacturing data, but can use any combination of BOM/routing codes).

Count Grade

A user-defined evaluation of a counting event.

Count Point

A predefined step on a routing or operation list where you can gather operation completion information. You define the appropriate points on the routing, record completions at these count points, and the system automatically backflushes the prior operations. This is only used on production IDs.

Counts

Count elements allow you to count the number of days or hours from a specific period of time. Counts are used primarily during proration calculations, but can potentially be utilized in other situations as well.

Court-Ordered Benefits Coverage

As prescribed in Title 5, United States Code and Title 5, Code of Federal Regulations, court orders that stipulate that an employee must continue or begin the coverage features for all employee benefits must be enforced. Federal employees are mandated by court orders to continue covering or begin covering their former spouses and/or children under their federal employee benefit programs (health, life, and thrift savings).

Court-Ordered Garnishments

As prescribed in Title 5, United States Code and Title 5, Code of Federal Regulations, court orders enforcing child support, alimony, or collection of commercial indebtedness are served on the appropriate entity within the Federal agency and implemented as offsets against the employee's salary.

Coverage

An employee's chosen benefit plan and coverage level; that is, what sort of benefit is provided as well as the value.

CPAM (Caisse Primaire d'Assurance Maladie)

In France, CPAMs are the local social security offices that manage health coverage for French workers. CPAMs are regulated and established by the French government. If you're managing a French workforce you'll need to identify and track the CPAM offices that impact your enterprise.

CRAM (Caisse Régionale d'Assurance Maladie)

In France, CRAM is the regional social security body which oversees the running of CPAMs. CRAM offices work with companies to both prevent and compensate workers for industrial injury.

Create Date

The date that you extracted a deduction or offset to PeopleSoft Deduction Management or created a split deduction.

Create Missing Items

In PeopleSoft Demand Planning and Inventory Planning, a feature that enables automatic system generation of master records that don't exist in the system.

Created Time

Time collecting device time or elapsed time generated by the system based on the time reporter's schedule (see Time Administration)

Creating Time

The preliminary generation of time segments as close as possible to their likely values when you officially report time—so that the information on the time records is as fresh and current as possible. The system shows you time that has already been created, rather than you having to create it "on the fly" when you come in to report. The process fills in reporting day gaps as defined by work schedules.

Credit Analyst

A required field used in PeopleSoft Receivables, Billing, Order Management, and Deduction Management when working with items. Each item must be assigned to a credit analyst. If no credit analyst is assigned to an item, the credit analyst assigned to the customer is used as the default.

Credit Risk Spreads

In the financial services industry, the additional charge to a risk-free interest rate, based on a riskier credit rating.

Credits

See Flexible Credits.

CREF

Acronym for Content Reference.

Crew Reporting

A Time and Labor process that enables you to report the earnings which consist of one or several time reporting codes and associated quantities of hours, amounts, or units, and task information for one date under report for a user-defined crew. The system transforms the information into instances of daily time for each crew member for the entered date.

Critical Success Factors (CSFs)

In PeopleSoft Balanced Scorecard, things that an organization must do well or excel at to achieve its goals. One or more key factors or objectives that must be accomplished for a particular strategic thrust. Key Performance Indicators are attached to CSFs.

CRM Warehouse

See Warehouses.

Cross Border Walker

This term is used in Europe for an employee who lives near a border in one country and works in another country. Such employees are subject to different tax and social security rules.

Cross-Plan Validation

The process by which the PeopleSoft Benefits Administration determines enrollment prerequisites for benefit plans. You can define four types of cross-plan validation prerequisites: prerequisites based on plan types, benefit plans, dependent enrollments, and coverage percentage limits for Life and AD/D plans.

Cross-View Reconciliation

In PeopleSoft Demand Planning, a process that enables the balancing of forecasts between selected levels of related views with the same **Forecast Item** key. The process is used when adjustments have been made to a working view and are then required in a related view.

Cube

See **Multidimensional Database** (MDDB).

Cube View

In PeopleSoft Demand Planning, defines the user's own view of a forecast. The parent working view and dimensions determine what forecast data is included and how aggregates are formed.

Cumulative Tax Method

A payroll tax calculation method that adds together year-to-date earnings and earnings for the current pay period, then annualizes the result before calculating tax. This method is useful when Payrolls vary greatly in amounts from pay period to pay period, such as in the case of sales commissions.

Currency Calendar

In the financial services industry, business calendars for markets outside the organization's domestic operations that reflect the foreign markets' holiday schedules.

Currency Conversion Engine

A PeopleSoft Enterprise Warehouse Engine that processes financial information in multiple currencies.

Current Period

The earliest pay period for which the close date has not passed (see Time Reporting).

Current Period (Time and Labor)

In Time and Labor, the employee's current pay period which will be determined via the employee's Pay Group affiliation. Although there can be only one definition of Current Period per installation, the user can change it manually.

Current View

A reporting screen in Time and Labor whose effective date is within the date boundaries of an employee's current pay period, and for which pay has not yet been confirmed. A *Future Time Reporting Transaction* is one that has an effective date after the last day of the employee's current pay period. An *Historical Time Reporting Transaction* is one that has an effective date before the first day of the employee's current pay period.

Current Year

A period for event maintenance processing.

Curve Generator

A supporting module (common to financial services industry applications) that enables you to construct curves used to determine appropriate interest rates for given maturities and / or time periods. You can import market data from outside sources such as Bloomberg, upload the data from a spreadsheet, or manually enter the data. You can then build configured curves from segments or combinations of other curves.

CUSIP Number

A nine digit alphanumeric number associated with issuers' securities. CUSIP (Committee on Uniform Securities Identification Procedures). A uniform numbering system widely used to identify specific securities and their issuers.

Custom Statement

A user-created logical or mathematical expression that determines information about an employee in PeopleSoft Pension Administration. Custom Statements commonly define employee groups and benefit formulas.

Customer Inquiry

A window containing options to review customer balances, aging, history, items, actions, and conversations.

Customer Scorecard

See PeopleSoft Customer Scorecard.

Customer Tree

A user-defined graphical representation of your current sales organization. A customer tree is used to establish and distribute funds and to determine authority levels for promotional activities.

Cut Session

Cut sessions are a means of dividing a course session. You use cut sessions where a course session does not run on consecutive days from start to finish, or if there are multiple instructors or locations. Each cut session has it's own start/end date, location, and instructor. For example, if you have a course that runs for two days a week for a month, you would divide the course session into four cut sessions, each of which is two days long.

Cycle Count

A manual counting event that does not cover an entire inventory business unit. Usually includes every item (and lot, if applicable) in a location or family.

Cycle Interval

The number of days between cycle counts.

Cycle Procedures

Inventory planning tasks that need to be performed on a regular basis to ensure an up-to-date **Inventory Policy**. The tasks can be performed either at the end of a processing period or within the period, and should always be performed if the forecast or **Control Group** or **Policy Item** parameters change. Tasks include generating a policy and reviewing **Work Queue** messages.

D

DAT file

A text file (input.dat) used with the Verity search engine that contains all of the information from documents that will be searchable but not returned in the results list.

Data Elements

Data elements, at their most simple level, define a subset of data and the rules by which to group it.

For PeopleSoft Balanced Scorecard, data elements are used as the basis for key performance indicators, and as target values for Key Performance Indicator (KPI) objects.

For Workforce Analytics, data elements are rules that tell the system what measures to retrieve about your workforce groups.

Data Entry Access List

Used to present a concise list of often-performed data entry tasks to a user. You can assign multiple control plans to a single data entry access list.

Data Extract

A report that creates a file used to transmit data to a third party on magnetic media. There is no meaningful printed output for this type of report.

Data Loader

Data Loader is a PeopleSoft Enterprise Warehouse utility that moves data from the Operational Data Store staging area to either the ODS reporting area or the Data Warehouse. The Data Loader utility is made up of several pages that allow you to enter Metadata to define your source and target records and your transformation rules and then perform the load by running an Application Engine.

Data Loader Map

Defines how to extract data from the Operational Data Store (ODS), transform it, and load to a Target Table. The target table can reside in the warehouse or the ODS layer.

Data Manager

A PeopleSoft Enterprise Warehouse engine that distributes revenue, expense, analytical application engine results, statistical quantities and other measures across business units, departments, products, customers and channels—any field or logical group in the chart of accounts. You can define a number of types and options within this engine. It is also used as a means of posting to the Performance Ledger.

Data Manager Rules

In the PeopleSoft Enterprise Warehouse, Data Manager rules use Constraints to specify the source as well as the target tables for moving, aggregating, or multidimensionalizing your engine output. Rules use Data Manager methods to enrich the PeopleSoft Enterprise Warehouse data.

See Data Manager Methods.

Data Manager Methods

There are several methods: Copy, GL Mapper, Prorata, Spread Even, and Tree Aggregation. Each method enables you to move and/or enrich engine output.

Data Mart

A Data Mart is a data structure that uses a central fact table and related dimension tables to generate a "relational cube" or directly generate an Insight report.

Data Mart Builder

The Data Mart Builder is a multiple Application Engine (AE) process, that is, a framework of procedural programs, that creates a Data Mart.

DataMaps

Information that builds upon the data captured in the TableMap records. DataMaps enable you to define a logical view of the physical PeopleSoft Enterprise Warehouse tables. DataMaps bring together information from many different tables and fields and define it all as one entity or table.

Data Row

Contains the entries for each field in a table. To identify each data row uniquely, the system uses a key consisting of one or more fields in the table.

DataSet

DataSets are used as input for various engines and processes, for instance, the Analytic Forecasting component, the Data Manager, user defined functions, drivers in Activity-Based Management, and data elements in the Key Performance Indicator Manager. DataSets provide a user defined set of information to the engines. DataSets use Constraints to restrict used columns and restrict returned rows. Each DataSet is created by a process specific setup. However, the underlying logic is the same, enabling you to more easily understand the functional aspects of the process.

Data Warehouse

A large database containing data summarized from one or more transactional systems, optimized to support the analysis needs of the enterprise. An ideal data warehouse contains all the data necessary to make business decisions. Users analyze the data in the warehouse using Online Analytical Processing (OLAP) tools and ad hoc query/reporting tools. An increasing

number of organizations have "virtual" data warehouses, where the data warehouse is not one physical database, but rather a collection of specialized (and distributed) data marts.

See also PeopleSoft Enterprise Warehouse.

Data Warehouse Tables

Data Warehouse tables act as the portal for getting data into the PeopleSoft Enterprise Warehouse from PeopleSoft, OLTP applications or other “outside” sources. These tables are used:

- As targets for loading operational data.
- For error detection and handling
- For data validation.
- For aggregation.

Database Alias

The PeopleSoft Pension Administration utility that looks up employee data.

Dataset

A file containing data to be analyzed by the Quality Server program. The dataset is similar in content to a spreadsheet.
In PeopleSoft Planning, a file that stores schedule information such as tasks, resources, calendars, and so on.

Date

See **Accounting Date Transaction Date** or **Effective Date**.

Date

If you want to either include a date in a calculation, or determine a new date by taking a starting date and either adding or subtracting a period of time to come up with another date, you use a date element.

Date Classified

Date the Position Description is approved by Management/Position Management.

Date Eligible To Retire

Date an employee is eligible to optionally retire based on the combination of age and service that meets legal requirements.

Date Under Report

The date (day) in PeopleSoft Time and Labor for which time is being reported. The Date Under Report does not have to equal today's date.

Day Breaker

Customer defined time that is used to determine when one day becomes the next. It's used to determine the "logical" date of a punch. (See Understanding Workgroups.)

Days Supply

In PeopleSoft Inventory Planning, a method that can be used with several types of **Inventory Policy**. Using this method, a specific number of days of supply for an item should be used to calculate the item's inventory policy.

Deal Type

PeopleSoft Treasury has categorized deals into several basic deal types from which you can choose when defining an instrument.

Death Coverage

The PeopleSoft Pension Administration function that determines the factor used to reduce an employee's benefit when the plan charges for PRSA coverage.

Decompressed Split

In PeopleSoft Demand Planning, a function for returning a compressed split database to its original form. See also Compressed Split.

Deduction

Any amount taken from an employee's pay check each pay period. Deductions may include health or medical benefits, union dues, and so on. See also Benefit Deduction and General Deduction.

Deduction Date

The as of date for the deduction item in PeopleSoft Receivables.

Deduction Item

An individual item that you created in receivables and is an open receivable on the customers account due to a deduction that they took in a payment for a receivable item.

Deduction Reason

Code that describes the type of deduction. When assigned to a write-off resolution, it determines what accounting entries to create.

Deduction Specialist

The individual responsible for tracking and resolving deductions in PeopleSoft Deduction Management.

Deduction Subset

A group of deductions selected from a company's standard set of deductions. Deduction subsets minimize data entry time in special processing situations such as bonus check runs.

Default Mode (DM) model

In the financial services industry, an approach used by financial institutions to predict a decline in portfolio value. Only two outcomes are considered – default or non-default. If the debt does not default, there is no change in the value. If the debt does default, then the loss is calculated as the difference between what was contractually owed and the value of any collateral recovered.

Defection Analysis

In PeopleSoft Workforce Analytics, the identification of employees who are likely to leave the organization based on predefined assessment criteria.

Deferred Compensation

Compensation payments that are payable to an individual in the future such as pension plan payments, annuities, stock awards and profit sharing. Note: Profit sharing can be considered direct pay if paid out in cash on a periodic basis or deferred pay if cumulative with the intention of payment in the long-term future.

Deferred Vesting

The adjustment made to the original option's vesting schedule that pushes the vesting into the future.

Defined Benefit Plan (DB Plan)

A retirement income plan (usually called a pension plan) where the employee's benefit is definitely determinable based on a plan-specified benefit formula.

Definition or Function Definition

The parameters for any of PeopleSoft Pension Administration's nineteen core functions. A definition has to be explicitly associated with an employee Group Definition before it can be applied.

Dekit

The ability to return material issued in kits to inventory. This is used when entire kits need to be returned; individual components are handled through kit issues/returns.

Delete Non-Matching Items

In PeopleSoft Inventory Planning, an option used in the Generation process to delete Inventory Planning items that don't have corresponding items in Demand Planning. The item deletion occurs when the system generates the policy.

Delta

When retroactive processing occurs for a given payee, the system recalculates each element generated for the payee. The system compares the recalculated results to the original results. The difference between these results is typically referred to as the retro "delta." A retro delta can represent either an underpayment or an overpayment that results in an adjustment to the payee's earnings.

Demand

Collection of training requests. This could be an employee demand, a departmental one or a company-wide demand.

Demand Filter Width

In PeopleSoft Demand Planning, specifies the confidence interval within which demand is considered to be reasonable. Actual demand that is outside the confidence interval is automatically filtered and replaced by the value at the edge of the interval. The value is expressed as a percentage.

Demand Filtering

In PeopleSoft Demand Planning, provides a way to detect and highlight unusual demands and forecast errors. If the demand falls outside of a band that is considered reasonable, the system automatically adjusts it to the level of the boundary and logs a message to the **Work Queue**.

Demand Number

The configured product sub-component sequence number.

Demand Planning

In PeopleSoft Activity-Based Management, this type of planning focuses on studying the impact of cost objects and activity volumes.

Demand Priority

The placing of importance on independent demand. The Planning engine uses the demand priority value to determine the order in which you fulfill the demand. You can set a demand priority from 1 to 998 with 1 being the most important level. The priority value of 999 is reserved for the system.

Demand Priority Rules

In PeopleSoft Inventory, a set of rules that will sort demand so the most important demand will have the first opportunity to reserve available inventory. If demand priority rules have

been defined, the Material Reservations process (INPLDMND) sequences orders by priority rank, processing those with the lowest rank value first.

Deplete Cost Method

Determines how you cost a depletions transaction within a book. The deplete cost methods available include Actual, Non-Cost, Perpetual Weighted Average, Periodic Weighted Average, and Value at Current Standard.

Depreciate When in Service

A switch that indicates whether PeopleSoft Asset Management should allocate depreciation as of the date an asset was placed in service. This is valid only in the year the asset was acquired.

Depreciation - Declining Balance

Budgeting calculates this as: Cost minus Accumulated Depreciation divided by Life divided by number of periods per year. It results in a higher depreciation expense in the early years of an asset, which decreases as you near the end of its useful life.

Depreciation - Double Declining Balance

Budgeting calculates this as: Cost minus Accumulated Depreciation multiplied by 2 divided by Life divided by number of periods per year. It results in a higher depreciation expense in the early years of an asset, which decreases as you near the end of its useful life.

Depreciation Methods

The various methods of spreading the aquisition cost across the life of an asset rather than expense the full value of an asset at the time you acquire it The value of the asset consequently decreases (or depreciates) through time. The four depreciation formulas delivered with PeopleSoft Budgeting include: declining balance, double declining balance, straight line, and sum of the years.

Depreciation - Straight Line

A method of depreciating asset value in equal amounts across the life of the asset. Per-Period Straight-Line depreciation is calculated as the cost of an item minus the salvage value divided by the number of periods to depreciate.

Depreciation - Sum of the Years

A depreciation method equal to the value of the remaining years of life divided by the sum of the years remaining is multiplied by the Net Book Value. This figure is then multiplied by the percent of years to depreciate. This results in a higher depreciation expense in the early years of an asset, which decreases as you near the end of its useful life.

Depromote

In PeopleSoft Demand Planning, the process of making an adjustment to actual demand data that removes the effect of a promotion during a defined period. As a promoted period moves into history, the system creates an adjusted demand entry that is equal to the **Prorated Forecast**.

DeptID

A ChartField that defines departments or administrative offices that have operational, fiscal and/or budgetary responsibility for specific sets of activities.

Derived Metric

The result of a calculation on a report of base metrics.

Detail

A temporary assignment to a different position for a specified period when the employee is expected to return to his/her regular duties at the end of the assignment. This employee is considered for pay and strength count purposes to be permanently occupying his/her regular position. Unless the agency chooses to use an SF50, a detail is documented with an SF52.

Detail Tree

A tree that employs ranges of detail values under each node; you must manually specify the detail values.

DFI ID (Depository Financial Institution ID)

A PeopleSoft Payables bank identifier, consisting of Transit Number, Swift ID, or CHIPS ID.

Dimension

A single element of a business model, such as product, department, or location. Cube Manager uses the term Conforming Dimension.

In terms of data analysis, dimensions can be thought of as criteria, such as time, product, and location, used to pinpoint a particular piece of data. For example, in the retail industry a set of dimensions could be geography, product, time, customer, and vendor. The geography dimension would include company, chain, region, district, and finally store attributes. A dimension is also a column heading on an analysis and reporting template which you can drill through or roll up to the multiple levels.

In PeopleSoft Budgeting, a view option that assists in summarizing the rows of data in line-item budgeting.

A single element of a budgeting model, such as account, product, project, department, or operating unit. In PeopleSoft Budgeting, these dimensions typically represent the ChartFields used by your organization during your budgeting process.

Dimension Table

In the PeopleSoft Enterprise Warehouse, Dimension Tables store additional attributes or data about Facts. Some example dimensions include Customer, Channel, Geography and Product.

Direct Compensation

In PeopleSoft Workforce Analytics, Direct Compensation is payment made to workers in exchange for their contributions to the organization. Direct Compensation is typically categorized as including Cash Compensation and Long-term Variable Compensation.

Cash payments made to workers in exchange for their contributions to the organization. Direct pay is typically categorized as fixed pay (for example, base pay, shift differentials) and variable pay (for example, profit sharing, incentive, bonus). Note: Profit sharing can be considered direct pay if paid out in cash on a periodic basis or deferred pay if cumulative with the intention of payment in the long-term future.

Direct Calculation

Calculate actual and directly assigned dollars.

Direct Cost

In PeopleSoft Workforce Analytics, a direct cost of an activity or a cost object. An example is the salary cost of employees working on a project.

Director

An affiliate of the company who holds a seat on the Board of Directors for the corporation. A Director, generally, is not an employee of the corporation.

Disability and Discrimination Act of 1995

In the United Kingdom this act makes it unlawful to discriminate against individuals on the basis of their disability in relation to recruitment, promotion, training, benefits, terms and conditions of employment, and dismissal.

Disability Rate Code

The desired percentage of disabled persons that should be employed by French employers, as mandated by the French government.

Disbursement View

In PeopleSoft Demand Planning, a **Forecast View** that allows the forecast from a working view to be reported on using an alternate key. Disbursement views are built directly from the working views and inherit many working view attributes, including time period and associated user data definitions, from the parent working view.

Discounted Stock Option

Rights to a stock option at a price less than 100 percent of fair market value at the time of grant.

Discretionary Plan

In PeopleSoft Workforce Analytics, this is a plan for distributing compensation awards that provide managers the ultimate discretion over a pool of money which is either funded based on company, group, or employee performance, or it's budgeted. The discretionary award determination is sometimes guided by a pre-determined percent of the participant's salary, expressed as an opportunity. This figure can then be modified based upon management's perception of actual value created by the group or employee.

Disqualifying Disposition (DD)

When an optionee sells or otherwise disposes of the shares of stock acquired through the exercise of an incentive stock option or through an employee stock purchase plan before the holding period for preferential tax treatment has lapsed.

In the case of Incentive Stock Options, the holding period is one year of the date of exercise and two years of the date of grant. At the time of disposition, the individual recognizes compensation income equal to the difference, if any, between the option price and the fair market value of the corporation's stock on the date of exercise. If the sale price is less than the fair market value of the stock on the date of exercise, the compensation income is limited to the total sales price less the total option price, less any fees.

In the case of purchases through an employee stock purchase plan, the holding period is one year from the purchase date and two years from the enrollment date. Compensation income in a disqualifying disposition is equal to the difference between the total fair market value on the purchase date and the total purchase price.

Distribution

Provide a repository of time and associated estimated and actual allocated labor costs to other systems

The process of assigning values to ChartFields. A distribution is a string of ChartField values assigned to items, payments, and budget amounts.

Distribution Network

A distribution network is a prioritized list of Inventory business units (IBUs). When a customer orders a product, the system uses this network to determine which warehouse the stock ships from.

Distribution Profile

A definition of ChartField distributions assigned for compensation costs. A distribution profile can be used to set up defaults for how the system should distribute costs associated with a position's salary, benefits, and earnings. PeopleSoft Budgeting-specific.

Distribution Rule

You use distribution rules to determine the order in which the system searches for matches against the distribution sets matrix when sales orders are entered.

Distribution Set

Distribution Sets assign account distribution information to combinations of defining elements used on sales orders.

Distribution Type

An identifier that defines one of the different transactions that move an item into or out of an inventory business unit. Distribution types are used to create debit and credit transactions to the general ledger via the Journal Generator.

Dividend

Distribution of earnings back to shareholders, prorated by the class of security and paid typically in the form of money or stock. The amount of a dividend is decided by the Board of Directors and is usually paid quarterly.

Document Management

The process through which a user has complete control of document version including the ability to view, query, and edit documents in a secure vault. Document management enables you to seamlessly perform online document queries and view documents directly, launching them from within PeopleSoft applications. You can associate pertinent documents with engineering change requests (ECR), engineering change orders (ECO), item revisions, bills of material, manufacturing and engineering routings, production component lists, and production operation lists.

Document Sequence Number

A value that the PeopleSoft system assigns to a document (such as an invoice, voucher, or journal) when you create a document for a business unit that you have enabled for document sequencing. The system determines the number by the values of the business unit, accounting date, and document type.

Document Sequencing

A flexible method that sequentially numbers the financial transactions (for example, bills, purchase orders, invoices, and payments) in your system for the purpose of statutory reporting and tracking of commercial transaction activity. Document sequencing requires that you classify all financial transactions into three transaction types—journal type, journal code, and document type—and that within each transaction type, all documents you enter are numbered sequentially. When you create a document (such as an invoice, voucher, or journal), the PeopleSoft system assigns a document sequence number to that document.

Document Type

The final level of three categories for defining a financial transaction (or document), necessary when using document sequencing. It represents the business purpose of a financial transaction, such as domestic customer invoice or customer credit memo. Document type is within one and only one journal code; journal code is within one and only one journal type. Document type is the only required category, because the values of the other two categories can be derived from document type.

Dollar Tolerance

In PeopleSoft Inventory, the acceptable cost difference between expected cycle count quantities and actual quantities counted. This value allows a margin of error for an item during cycle count reconciliation based on item cost.

Domestic Relations Order (DRO)

A preliminary version of a court order (usually stemming from a divorce settlement) ordering a division of a participant's pension benefits. The order is not in effect until it is determined to be "qualified" by virtue of meeting certain requirement. At that point it becomes a Qualified Domestic Relations Order, or QDRO.

Double Byte Characters

If you're 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].

Draft Worksheet

A work space used in PeopleSoft Receivables to track a draft through its processing life cycle.

Drill-Back Calculation

Assigns indirect dollars and Drill-Back calculations. Also, this picks-up all costs in the Calculations Detail (CALC_DETAIL_F00) that was assigned during direct calculations.

Drill Down

The ability to go down to the next level of detail in a set of data. For instance, if you're looking at an expense figure for a division, you can drill down to the expenses for each department in the division.

Drill Down

The ability to go down to the next level of detail in a set of data. For instance, if you're looking at an expense figure for a division, you can drill down to the expenses for each department in the division.

Driver Lookup Table

Tables associated with a driver that enable different rates and amounts unique to a budget center.

Drivers

In PeopleSoft Activity Based Management, drivers are a means of assigning dollar amounts from resources, activities, and cost objects to other resources, activities, and cost objects throughout the model in PeopleSoft Activity-Based Management. Drivers can also be assigned across business units. There are different types of driver categories (transactional, duration, and intensity), and different ways of specifying how those dollar amounts are calculated (amount, percentage, spread even, and direct), as well as different ways that dollar amounts are assigned (depending on assignment type and object type).

In PeopleSoft Business Planning, a driver can be defined as a set of values that are used as an input to another process. In this context, a driver should be interpreted as a projection of external factors and other indicators. The user can the define a relationship between the driver and a financial result. The driver values and the defined relationship then combine to produce a projection of the financial result. For example, a projection of the number of square feet used (driver) and the price per square foot paid in rent (driver) can combine to produce a projection of rent expense (financial result).

DRO

See Domestic Relations Order.

DSS (Decision Support System)

A DSS is a workstation-based analysis and reporting system, typically aimed at analysts and line managers. OLAP tools provide a powerful DSS.

Duration

In PeopleSoft Pension Administration, the utility that calculates the length of time between two dates.

Duration [Global Payroll]

An element type that calculates a period of time between two dates. For example, if you want to determine a payee's age, you can calculate the duration between his birth date and the calendar period end date.

Dynamic Group

A group in Time and Labor that enables you to establish criteria or attributes for a group of employees. All employees who fit this criteria at processing time belong to the group.

Dynamic Tree

A tree that takes its detail values—*Dynamic Details*—directly from a table in the database, rather than from a range of values entered by the user.

Dynamic Views

In PeopleSoft Demand Planning, a **Forecast View** that allows interaction with the forecast data using an alternate key structure. By using dynamic views, you streamline the working view and can complete the working-view design without having to anticipate all conceivable adjustments.

E

Earliest Change Date

Determines both the range of dates and the amount of data that will be processed for each time reporter (see Batch Processing)

Early Punch

A punch that is more than the predefined number of hours/minutes before a scheduled punch where a time reporter is warned

Early Retirement Date (ERD)

A retirement date earlier than a plan-specified “normal” retirement date. Employees usually must meet age and/or service requirements to be eligible for early retirement, and early retirement benefits are often reduced to compensate for the longer duration of payments.

Early Retirement Factor

The reduction made to an employee's benefit if the employee elects for early retirement.

Early Warning

In commitment control, warning of possible future budget exceptions. You can specify that you are to receive a warning when commitments and expenditures reach a predetermined percentage of budget. For example, you can instruct the system to let you know when commitments and expenditures reach 50%, 80%, or some other percent of a budget.

Early/Late Adjustments

The PeopleSoft Pension Administration function that calculates early retirement factors or late retirement factors.

Earning Group

Part of a group of defaults assigned to job codes. Earnings group may include non-salaried items such as holidays and bonus pay dependent on individual company parameters.

Earnings

The amount owed to an employee based on salary, hours worked, or other calculation routines, plus other types of compensation and holiday, vacation, and bonus pay.

Earnings [Global Payroll]

An element type that defines the different types of compensation that are added to a person's pay. Examples include salary, commission, bonuses, and retirement pay.

Earnings Accrual Class

Categorizes a set of accruable earnings.

Earnings Code

Codes that represent the various types of earnings such as regular, overtime or leave.

Earnings Per Share (EPS)

The portion of a company's profit allocated to each outstanding share of common stock. Net income (reported or estimated) for a period of time is divided by the total number of shares outstanding during that period.

Earnings Type

An abbreviated and encrypted set of business instructions containing compensation instructions. Earnings Type may also contain Benefit Entitlement and Administration instructions, taxation instructions, Financial Accounting instructions, Organizational Administration instructions, work group and labor affiliation instructions, and other instructions.

Economic Loss

In Funds Transfer Pricing, this refers to the break fund economic loss, calculated by applying the theoretical value of the interest rate differential (IRD) against a cash flow stream, based on the amount of the prepayment or cancelled draw-down.

Economic Value Added

In the financial services industry, Economic Value Added is a financial metric that factors into the measurement of an activity's profitability the cost of economic capital assigned to that activity.

EDGAR (Electronic Data Gathering, Analysis, and Retrieval)

An electronic system implemented by the SEC that enables companies to file documents in conjunction with disclosure requirements mandated by the SEC.

EDI Agent

Used in EDI processing, the inbound EDI Agent loads trading partner data (flat files) into the PeopleSoft database using transaction, map, and trading partner definitions set up using EDI Manager. The outbound EDI Agent extracts information from the PeopleSoft database and generates data files that can then be processed for transmission to a trading partner.

EDI Manager

A suite of online pages used to define transaction sets, trading partner profiles, and translation maps for EDI transactions.

Edit Table

A table on 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.

EEO Company Code

In the United States companies are assigned this federal code for EE0 and VETS100 reporting.

Effective Date

A method of dating information in your system. 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.

Effective Date

A method of dating information in your system. 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. The Effective Date usually defaults to your system's current date.

Effective Periods

In PeopleSoft Demand Planning, the number of periods of historical demand used in the Model Reset process. The value can be used to exclude older, possibly unrepresentative historical demand data from model **Optimization**.

Effective Sequence

A system-generated number assigned to distinguish between two job entries with the same effective date.

Effective Tax Rate

The ratio of income tax paid over gross income, showing the percentage of income actually paid in taxes.

Effectivity Date

The date on which a component can be added or substituted in production, typically specified on an item's bill of material.

EIS (Executive Information System)

An EIS is a workstation-based analysis and reporting system for executives. An EIS provides a higher-level view of the data than a DSS, and typically requires less knowledge about the underlying transactional systems. OLAP tools provide a powerful EIS.

Elapsed Schedule

A method of scheduling a time reporter's time that is based on TRC and duration. This method can be used for scheduling of elapsed time reporters (see Scheduling.)

Elapsed Time

Reporting non-clock time in increments of hours or partial hours (see Managing Time / Understanding Time Reporting).

Elapsed Time Service

A method of calculating a period of service that uses only the start and end dates of the period to determine the amount of service. Hours worked or other measures of the actual work performed during the period are not taken into account.

Electronic Certification System (ECS)

An automated Payment Voucher authorized by the Certifying Officer for use within the Treasury Department, Financial Management Service's financial system. PeopleSoft provides a method to record and generate data files for on- and off-cycle processed payments.

Element

In PeopleSoft Global Payroll, an element refers to both primary elements and supporting elements. Primary elements are comprised of earnings, deductions, absence entitlements, and absence take elements. Supporting elements are element components that are combined to create primary elements.

In PeopleSoft Enterprise Performance Management, elements are used to create a Profile. An Element can be one or more columns of data in an Enterprise Warehouse table, associated with a single dimension (for example, Customer, Product, Department, or Channel). An Element can also be KPI, Population, subscription data from a third party, preexisting Profiles, and data mining scores.

Element Group

Element Group identifies a group of elements to provide eligibility. You can then use this as a notational shortcut—instead of having to list each element, you can use the element group name. Element Group's expedite the process of manipulating earnings and deductions.

Element Name

Name assigned by the user for data fields, rules, formulas, and tables. For example, the names you give to new rules, elements, or objects.

Element Segment

When an element changes mid-period, requiring the affected element (and perhaps a subset of other elements) to be calculated multiple times on either side of the date on which the change takes place, element segmentation is used. Unlike period segmentation, the system segments only the elements you select, and creates separate result columns only for the specified elements. In element segmentation, there is only one gross-to-net result set.

Eligibility Group

Eligibility groups define the possible earnings, deduction, absence entitlement, and absence take elements that a payee might be eligible to receive. This enables you to group payees so as to assign eligibility for certain pay elements.

Eligibility Rule

PeopleSoft Benefits Administration uses eligibility rules during Benefits Administration processing to determine which benefit programs and options an employee is eligible for. Eligibility rules are closely associated with event rules: they determine what options an employee can *have,* while event rules determine which of those options an employee will actually be able to *choose.*

Elimination Set

See **Consolidations-Elimination Set**.

Email Template

Pre-defined parameters that establish automatic email generation during budget submittal, rejection, publishing, and/or targeting.

Employee

An individual employed by an organization and administered as an employee in the PeopleSoft Human Resources system.

Employee Accounts

The PeopleSoft Pension Administration function that tracks employee contributions to a pension plan.

Employee ID

A unique identification code for an individual associated with your organization.

Employee Paid Benefit

The portion of a pension benefit funded by the employee's own contributions to the pension plan. Also, the PeopleSoft Pension Administration function that determines this amount.

Employee Profile

This PeopleSoft Activity-Based Management feature enables time and labor information to be part of an analysis.

Employee Stock Purchase Plan

A type of statutory stock option plan through which employers grant options to their employees in order to provide them with additional forms of compensation.

Employee Survey

In PeopleSoft Workforce Analytics, a method for capturing information about the activities performed by a given employee as well as the amount of time they spend performing each activity to perform activity-based management.

Employee Survey Report

The Employee Survey Report is an annual regulatory report that the French government requires from employers with more than 200 employees. In French it is called "Le Bilan Social". The report is communicated to both labor unions and the government. It provides a snapshot view of the company over the past 3 years for about 200 indicators.

Employee Training Cost

Amount budgeted to pay for students' salaries while on training courses.

Employer Identification Number (EIN)

In the United States a company is typically defined as a business enterprise that has a unique federal Employer Identification Number (EIN) for payroll tax reporting purposes.

Employer's Liability Insurance Associations (Berufsgenossenschaften)

Social Insurance in Germany is maintained and administered by private organizations that act as employer's liability insurance associations. Employers pay out premiums to these associations, who administer and pay out funds to workers who are injured on the job.

Employment Cost Index (ECI) Adjustment

Annual increase to wages established/permitted by statute.

Employment Equity Computerized Reporting System (EECRS)

Canadian companies are required to report to the Federal Government on employment equity. PeopleSoft Human Resources contains the Canadian Employment Equity report (PER101CN), which creates a data interface file to the federal government's Employment Equity Computerized Reporting System (EECRS).

Employment Record Number (EMPL RCD#)

A field in PeopleSoft Human Resources Management Systems and PeopleSoft Workforce Analytics that indicates an employee has multiple job records in the system. A numeric value (0, 1, 2) is assigned to each job as a way to uniquely identify that job record.

Encumbrance

A claim against funds. It is a projection of future expenses based on the situation, as we know it today. Encumbering funds is not the same as spending them or even guaranteeing that you will spend them. It just means that if the situation as it exists today does not change, you will spend all of those funds by the end of the fiscal year.

Engineering Bill of Material (EBOM)

A listing of all the parts, raw materials, and subassemblies that form the basis of all item and product structures. EBOMs differ from MBOMs (Manufacturing Bills of Material) in that they are not visible within Production Planning or Production Management and are isolated from Manufacturing.

Engineering Change Order (ECO)

A revision to a blueprint or design, released by engineering to modify or correct a part and/or bill of material. PeopleSoft Engineering uses ECOs to manage and document required assembly and component changes.

Engineering Change Request (ECR)

A document that allows you to request manufacturing process improvements and report product defects directly to the engineering department. When workflow is enabled, ECRs can also be routed for review and approval, after which they change into ECOs.

Engineering Cost Version

The process of generating cost versions for new and modified configurations based on engineering bills of material (EBOM) and costing data.

Engineering Workbench

An engineering environment, separate from production, consisting of engineering bills of material (EBOM), engineering change requests, engineering change orders, EBOM cost roll-up capability, online BOM comparisons, and seamless integration to a document management vault.

Engineering Workbench

An engineering environment, separate from production, consisting of engineering bills of material (EBOM), engineering routings, engineering change requests, engineering change orders, EBOM cost roll-up capability, online BOM comparisons, and seamless integration to a document management vault.

Enterprise

In PeopleSoft Time and Labor, all of the business units of the installation site.

Enterprise Performance Management (EPM)

See PeopleSoft Enterprise Performance Management

Enterprise Portal

The PeopleSoft Enterprise Portal is a separate product offering purchased independently of any other PeopleSoft applications. It can be used with or without any PeopleSoft application. It can be used as a standalone corporate portal that does not access PeopleSoft data at all.

Enterprise Resource Planning (ERP)

The encompassing term for all the transaction-oriented database applications an organization deploys across it's business enterprise. The term includes financial, manufacturing and supply chain, human resources, and payroll applications, among others.

Enterprise Warehouse (EW)

See PeopleSoft Enterprise Warehouse

Entry Authority

Authorization granted by employees to specific user IDs for entering expense data on their behalf.

Entry Currency

The currency used to enter budget data.

Entry Event

An automated process that generates multiple debits and credits resulting from single transactions, to produce standard supplemental accounting entries.

Entry Event Code

Designation of an Entry Event; an identifier or label.

Entry Event Generator

A mechanism that generates standard, supplemental accounting entries based on Entry Event codes.

Entry Event Process

An accounting transaction. Entry Event processes combine to form Entry Events. For example, requisition posting is a Purchasing process, and cash clearing is a Payables process. Each process can involve one or several Entry Event Steps.

Entry Event Step

Part of an accounting transaction. For example, the BUDG process includes these steps, among others: prepare allotment budgets, prepare organization budgets, and prepare revenue estimates. Entry Event steps combine to form Entry Event processes.

Entry On Duty Date (EOD)

Date that indicates when an employee started to work at his/her current agency.

Entry Type

Any activity that creates or updates an item.

EPM (Enterprise Performance Management)

See PeopleSoft Enterprise Performance Management

Equal Employment Opportunity Commission (EEOC)

In the United States the EEOC requires that most companies file one or more reports from a series named EEO-1 through EEO-9. These reports include counts by federal employment categories of male and female employees in certain ethnic groups.

Equitization

A process that enables parent companies to calculate the net income of subsidiaries on a monthly basis and adjust that amount to increase the investment amount and equity income amount before performing consolidations. For organizations with complicated parent/subsidiary business unit relationships, this automated process reduces the process time and reduces the possibility for errors.

Equity Increase

In PeopleSoft Workforce Analytics, Equity Increases are base pay increases granted to bring an employee's pay up to some internally specified standard for your organization.

Equivalent Standard Deviation

In PeopleSoft Demand Planning, a **Standard Deviation** developed during the Model Reset process that enables you to compare standard deviations from different models. The deviation is calculated by multiplying the **Model Equivalency Factors** defined on the **Control Group** by the standard deviation.

ERISA (Employee Retirement Income Security Act of 1974)

The U.S. Federal legislation enacted to prevent abuses of employee pension rights by employers.

Error Exception

A transaction that is stopped because the budget limits would be exceeded if it continued. For the transaction to proceed, action must be taken, such as canceling or reducing the transaction amount, increasing the budget amount, overriding the budget limits, or transferring available funds from another budget.

Error Ratio

The ratio of the statistical Standard Deviation to the base component that gives an indication of the accuracy of the forecast. The ratio is presented in the PeopleSoft Demand Planning Audit and Accuracy Review and is calculated during the **Forecast Calculation Process**.

Estimated Gross

Estimated labor cost associated with reported time (see Managing Time, Understanding time Reporting Codes).

Estimated Shipments

A group of shipment schedules used to manage sales order requested shipment data and actual shipment data. Used in conjunction with weight and volume pricing and freight charge calculations.

Ethnic Code

The Federal Office of Management and Budget (OMB) racial and ethnic census categories used for classifying individuals in U.S. Government reports.

ETL (Extract-Transform-Load)

See Extract-Transform-Load.

ETL maps

ETL maps provide rules for importing your source data to the data warehouse tables.

Evaluated Receipts Settlement (ERS)

A PeopleSoft Payables feature that matches receipts against purchase orders and generates vouchers without requiring an invoice.

Evaluations Periods

In PeopleSoft Demand Planning, indicates the number of future periods to use for the calculation of forecast errors. For example, if the number of evaluation periods is two, then the forecast error in April 2001 (after posting demand for April) would be determined by comparing the actual demand for April and the April forecast generated in February 2001. Evaluation periods are set separately for each view.

Event

Events are predefined points either in the Application Processor flow or in the program flow. As each point is encountered, the event fires on each component, triggering any PeopleCode program associated with that component and that event. Examples of events are FieldChange, SavePreChange, OnRouteSubscription, and so on.

Event

Occurrence or happening.

Event Class

An event or type of event that results in a change of benefits eligibility for an employee or dependent. Event classes are prominently used in COBRA and Benefits Administration processing.

Event Maintenance

The process that enables you to manage ongoing enrollments during a plan year. Changes involving maintenance include new hires and re-hires, terminations, family status changes, and changes to benefits eligibility.

Event Rule

Used by PeopleSoft Benefits Administration to determine how events are processed by the system. Event rules look at the benefit plan options an employee is eligible for and determine which options the employee can actually *choose.* Event rules are closely associated with eligibility rules but it is important to note that they are not the same. Event rules *should not* be used to determine eligibility.

Event Trigger

You use triggers to tell the system that when a change takes place to certain data (an event), it should perform an action automatically. When the event occurs, the system writes a line to a trigger table. Then when it's time for the action, the system reads the data from the trigger table and performs the appropriate action.

EW (Enterprise Warehouse)

See PeopleSoft Enterprise Warehouse.

See also PeopleSoft Enterprise Performance Management (EPM).

Expected Losses

In the financial services industry, the amount the institution predicts it will lose in portfolio value. Loan loss reserves are set aside to cover the expected losses.

Excepted Service

As defined by 5 USC 2103, the Excepted Service consists of those civil service positions that are not in the competitive service or Senior Executive Service.

Exception

User or system delivered, defined conditions applied to scheduled, reported or payable time that require audit or review (see Time Management)

Exception Rules

A rule (s) that is applied to scheduled, reported time, and payable time in order to determine conditions which require audit or review (see Time Administration).

Exception Severity

The degree of importance associated with an exception. For example, in exception which is a result of an employee clocking in late may have a Medium severity, while an exception which is a result of an employee not clocking in has a High severity (see Time Management).

Exception Time Reporting

A method of time reporting where only differences to the schedule are provided (see Time Reporting).

Excess Plan

A pension plan where the benefit formula provides an increased benefit for Final Average Earnings above a specified integration level. This compensates for the fact that Social Security benefits are based only on earnings up to a specified maximum.

Exchange Rate Variance

In PeopleSoft Cost Management, the change in currency exchange rate between the time the item is received into inventory and vouchered in Accounts Payable.

In PeopleSoft Payables, a matching feature that compares the exchange rate on the purchase order and the invoice and then copies any variance to PeopleSoft Inventory tables for analysis and accounting purposes.

Exclusive Pricing

Supersedes all pricing structures in effect for customers and products, except **Buying Agreement**, and enables you to drive pricing with a promotional structure. Exclusive pricing can be set up for a specific time frame and associated with particular orders.

Executive Schedule (EX)

Compensation and pay plan used by the Executive Branch of the federal government. Statutory pay limits are derived from several of the pay levels within this plan and imposed on the General Schedule and other existing pay plans throughout the Federal government.

Exercisable

The option shares that are available to the optionee to exercise.

Exercise

The transaction in which an individual purchases or "exercises" the right to purchase the option shares. The IRS refers to the purchase of company stock in an employee stock purchase plan as an exercise.

Exercise Date

The date on which an individual purchases underlying shares from and option grant or transacts a simultaneous purchase and sale of underlying option shares through a cashless exercise and collects option profit in cash or shares.

Exercise Price

The price per share required to exercise a stock option.

Exercise Proceeds

Cash, stock or other recognition received by a company as a result of option exercises, including cash or stock paid by individuals to exercise options and cash company tax savings from deducting non-statutory option profits at exercise.

Expense Location

Geographic area defined to enable the recording, tracking, and reporting of expense activity.

Expense Location Amount

Authorized spending defined for an expense type in a particular expense location and currency.

Expense Location Group

Collection of expense locations based on a common classification such as state, country, or continent.

Expense Type

Means of itemizing various kinds of business expenses. Examples are hotel, dinner, or ground transportation.

Expense Type Edit

User-defined requirement that mandates input of additional data—such as an airline ticket number or number of nights in a hotel—when an expense type is selected in an expense report.

Expense Type Group

Expense types that are classified together for reporting and tracking.

Expensed Item

Non-inventory item which may represent software, manuals, documentation, or any item for which no quantity on hand is maintained, but which can be specified on a bill of material (BOM). Expensed items can only exist as components on a BOM and cannot have a BOM, routing, or production area/item definition.

Expiration

The process by which the outstanding shares of an option cease to be exercisable, generally at the end of the option term. The length of the option term and the date of expiration are established in the Grant Agreement.

Expiration Date

In PeopleSoft Inventory, the date a lot exceeds its Shelf Life and is no longer acceptable for fulfillment or consumption. (Expiration Date = Creation Date + Shelf Life)

Expiration Date

The last day of an option term in which the option is canceled and no longer exercisable.

Expiration Grace Period

When you enter a stock action allows the exercise of the already vested shares as of the action date, the system will calculate the date these shares expire based on the grace period defined on the Stock Action Rules page for that stock action. The system will automatically cancel vested shares not exercised at the end of the expiration grace period.

Express Customer

A customer for whom the minimum necessary information is entered.

Expressions

Expressions enable you to create pseudo-columns made up of mathematical calculations based on actual fields on a table. Since expressions are resolved at run-time, duplicate information is not stored on the database.

Express Order

An order entry shortcut in PeopleSoft eStore and Mobile Order Management whereby the customer populates the shopping cart and goes directly to the order summary to checkout, bypassing any billing or shipment modification screens. Billing and shipping information defaults in as previously entered.

External Data

Data from external sources. For instance, in PeopleSoft Workforce Analytics, external data may include third party salary surveys and benchmark metric surveys.

External Scheme

In the United Kingdom an External Scheme is a vocational training, education and job placement program involving an employee, an employer and the government.

External System

Any system that is not directly compiled with the PeopleTools servers.

Extra Time

Any hours worked outside of an employee's normal (scheduled/shift) hours or days. Extra time may be scheduled in advance of when it is worked, and may be subject to special compensation rules. It may be treated differently than standard time for purposes of Benefit Entitlement and Administration.

Extraction

A reusable query that specifies what information should be retrieved from the Quality database.

Extract-Transform-Load (ETL)

The extraction and transport of data from one server to another remote server. In PeopleSoft budgeting ETL specifically refers to the process by which financial and human resource data is extracted from PeopleSoft Financials and HRMS and transferred to the PeopleSoft Enterprise Warehouse which PeopleSoft Budgeting uses to access and record data transactions. Within PeopleSoft Enterprise Warehouse, data migration typically refers to information moved from outside sources into the Operational Data Store tables.

Extrinsic Rewards

Tangible rewards that can be given to the individual. Typically categorized as financial and non-financial rewards. Financial rewards would include direct compensation, indirect compensation and deferred compensation. Non-financial rewards are provided to the individually and viewed as a benefit by the individual based on the culture of the organization such as the size or location of one’s office.

In PeopleSoft Workforce Analytics, tangible rewards given to an individual. Typically categorized as financial and non-financial rewards. Financial rewards would include direct compensation, indirect compensation and deferred compensation. Non-financial rewards are provided to the individually and viewed as a benefit by the individual based on the culture of the organization such as the size or location of one’s office.

F

Fact

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.

Fact Table

A fact table is where facts are stored in the PeopleSoft Enterprise Warehouse.

Fair Labor Standards Act (FLSA)

A federal regulation governing several time and labor issues. *FLSA Overtime* requires that all nonexempt employees be paid at a rate of time-and-one-half for all hours over 40 physically worked during a workweek. This requirement may be superseded by state or local laws when the lesser law is to the greater benefit of the employee, or by union contract. An *FSLA Workweek* is a permanently established, regular workweek for a group of employees.

Fair Market Value (FMV)

The price of a company stock based on the current market value as determined by supply and demand, or a valuation method. The stock market sets the fair market value for a public company. For a private company the fair market value is more subjective, but typically determined by financial factors or set by an outside valuation company.

Fair Market Value Tracking Methods

Methods used to track and report trading activity on various exchanges (i.e. NYSE, AMEX, NASDAQ, etc…).

Family Medical Leave Act (FMLA)

A federal regulation that protects health benefits and job restoration for employees who must take a leave from work to care for themselves or family members. FMLA regulations contain provisions regarding employer coverage, employee eligibility and entitlement, notice and certification, continuation of health benefits, and job restoration. PeopleSoft Benefits applications offer FMLA Plans that help employers and employees determine FMLA eligibility and schedule and track FMLA leave requests.

Federal Employee Group Life Insurance Program (FEGLI)

Generally, if the employee has Federal retirement coverage or is on a temporary appointment exceeding one year, he/she is eligible to participate in the FEGLI program. Once eligible, he/she is covered automatically for Basic Life Insurance and premiums will be deducted from gross salary unless coverage is waived within the first period of eligibility. The program offers Basic Insurance coverage and three types of optional coverage: Option A (Standard), Option B (Additional), and Option C (Family).

Federal Employee Pay Comparability Act (FEPCA)

This law provides a structure and methodology to determine and authorize locality-based pay adjustments to Federal employees in order to elevate their basic pay to be commensurate with private sector employees working in the same occupations in the same geographic localities. It also includes a feature to authorize agencies to make advance salary payments to attract candidates for open positions which have consistently been hard-to-fill in certain geographic areas.

Federal Employees' Compensation Act (FECA)

This law provides compensation and medical benefits to civilian employees of the United States for disability due to personal injury or disease sustained while in the performance of duty. A feature of this law provides for the continuation of pay (COP) without charge to leave for up to 45 calendar days due to disability and/or medical treatment following a traumatic injury. Employees file claims with the U.S. Department of Labor, Office of Worker's Compensation, which adjudicates the claims and compensates the employing agencies for the employee's pay and benefits during the claim period.

Federal Employees Health Benefits (FEHB)

Generally, the employee is entitled to coverage by the FEHB program if appointed to a position with Federal retirement coverage or has been on the rolls on a temporary appointment for more than one year. The Federal employer shares the cost of the premium (about 75%); actual premiums depend on the plan selected. If under a temporary appointment, the employee pays both the employer and employee shares. If the position is part-time, the employee pays the employee share and a portion of the employer’s share.

Federal Employees Retirement System (FERS)

A retirement plan available to employees of the federal government. FERS covers all employees appointed to a position in the federal government after January 1, 1987. Coverage includes Social Security, a basic annuity plan, and a TSP.

Federal Employer Identification Number (EIN)

Used to identify the tax accounts of businesses. Businesses, which have employees or operate business as a partnership or corporation, must obtain an EIN.

Federal Insurance Compensation Act (FICA)

Employee and employer contributions to Social Security.

Federal Reserve Transit Number

A unique identifier for U.S.-based banks, allowing banks to transfer funds within the Federal Reserve system.

Feeder Line

A type of production line replenishment used in PeopleSoft Flow Production. If you are using feeder line replenishment, smaller production lines create subassemblies that feed directly to your production line.

FEGLI Living Benefits Act

Beginning 7/25/95, a Federal employee who is terminally ill may elect to receive a lump-sum payment equal to the full amount of basic life insurance only, or a limited portion designated in multiples of $1000. An election to receive this benefit is irrevocable; the individual is considered terminally ill if his /her life expectancy is 9 months or less.

FICA (Federal Insurance Contributions Act)

FICA consists of both a Social Security (retirement) payroll tax and a Medicare (hospital insurance) tax. The tax is levied on employers, employees, and certain self-employed individuals.

Fictitious Calculations

Fictitious calculation rules perform temporary calculations. A fictitious calculation is a sub-calculation run during a normal calculation to determine a net that would have been computed if certain parameters were used. This result is then used for further processing in the normal calculation. A fictitious calculation is always started from inside a normal calculation, run for one payee, and run for a specified set of periods.

FIFO (First In First Out)

Method used by companies to record Disqualifying Disposition Income. If a company uses this method they record the optionees disposition of shares by attributing the shares to the earliest exercise, purchase or release dates for which shares remain available for sale.

Fill-In Employment

Employment held by persons during the time period after leaving their regular occupation in anticipation of, but before entering, military service.

Filter

A filter creates a subset of information. Filters are used in templates to limit your information from a pick list of attribute values.

Final Average Earnings (FAE)

The PeopleSoft Pension Administration function that averages earnings from a specified period of an employee's career. The result is used as a component of the pension benefit formula.

Final Forecast

The final forecast is the prorated version of the adjusted forecast, summarized to all levels of the product hierarchy. This is the best-guess version of the forecast that is used to make all decisions dependent on the forecast.

Final Table Merge Engine

Final Table Merge Engine is used by the PeopleSoft Enterprise Warehouse; it moves enriched data from one table into another. When you run a job in a jobstream, the immediate results are stored in temporary tables. At the end of the jobstream, the Merge Engine runs and merges the output temporary tables into the final tables, where processing can continue.

Financial Accounting

The accounting for a business entity's assets, liabilities, revenues, and expenses to determine its net worth and to produce financial statements. Within Generally Accepted Accounting Principles, a business has some latitude as to when and how to record its financial transactions, as long as it continues to meet its legal and regulatory requirements. A business' financial accounting requirements are not necessarily the same as its cost accounting requirements. The one should not be mistaken for the other (i.e. the extent to which a company's financial accounting system meets its cost accounting needs depends on how it has chosen to describe its chart of accounts and the level at which it has chosen to record financial transactions.

Financial Instrument

In the financial services industry, a specific product or service sold by a financial institution to its customers. In terms of the reporting hierarchy, a product falls under a ledger account, while an instrument falls under a product. A product may be treated like a generic description or category, while an instrument is a specific instance of a category.

Financial Performance Measures (FPM)

For the financial services industry, the Financial Performance Measures program performs calculations on financial instruments based on the rules defined in the Financial Calculation Rules module, and using input from the Cash Flow Generator, Stratification engine, Product Pricing, and Curve Generator. Its calculations include: measures of duration, option-adjusted spread and option-adjusted cost for PeopleSoft Funds Transfer Pricing, and Monte Carlo simulation for PeopleSoft Asset Liability Management.

Financial Product

In the financial services industry, a product or service sold by a financial institution to its customers. In terms of the reporting hierarchy, a product falls under a ledger account, while an instrument falls under a product. A product may be treated like a generic description or category, while an instrument is a specific instance of a category.

Financial Services Instrument

In the financial services industry, products created by financial institutions and sold to retail customers. Product prices and interest rates are set by the financial institutions and take into account its customers' behavioral models.

Financial Statement Simulation

A facility within Planning & Simulation which establishes rules for simulating future period, or pro-forma, financial statements. The user defines corporate financial policies, such as corporate tax rates, dividend distribution frequency, and force balancing rules, which are then applied to cash flows for a given future accounting period. The Financial Statement Simulator engine drives costs and revenues to accounts on PF_LEDGER_F00 via a scenario.

Financials Warehouse

See Warehouses.

First Year Amount

See 1st Year Amount.

Fixed Basis

The basis option enables you to create the data for the Basis online, as part of the rule. Fixed Basis is used with the Allocation Manager only. It is available with all methods except when Period-Based Allocation is being used. The Fixed Basis is a predetermined table that can be populated online.

Fixed Offering

The offering type is fixed when the end date of each offering is the same for all employees regardless of the employee's grant dates.

Fixed Percentage

A fixed percentage value. The source pool amount will be split based on this percentage to get the target amount. Used with the Allocation Manager.

Fixed Period Requirements

In PeopleSoft Enterprise Planning and Production Planning, a lot-sizing technique that sets the order quantity to the demand for a given length of time.

Fixed Picking Bin

A dedicated picking location for an inventory item. Fixed picking bins are replenished from bulk locations when the available quantity falls below the optimal quantity.

Fixed Plan

A stock purchase offering period where the ending offering date will be the same as the purchase date. Eligible employees will always purchase stock on the specific purchase dates and by the purchase rules you define.

Fixed Quantity

An **Inventory Policy** method that defines a fixed amount of an item to be ordered to meet replenishment needs. This method can be selected as an inventory policy for order quantity, safety stock, **Reorder Point**, and minimum and maximum parameters.

Fixed Source

The fixed source option enables you to create the data for the Source online, as part of the rule. Fixed Source is used with the Allocation Manager only. It is available with all methods except when Period-Based Allocation is being used. The Fixed Source is a predetermined table that can be populated online.

Flexible Credit

Any credit associated with a given benefits program, plan, or type of coverage. Credits based on an entire program can be applied toward the benefit costs however the employee chooses.

Flexible Hours

Hours during the workday, workweek or pay period during which a time reporter covered by a flexible work schedule may choose to vary his times of arrival and departure from the worksite (see Scheduling)

Flexible Spending Account (FSA)

An account to which an employee and (optionally) an employer pledge an annual amount for a plan year. The employee then submits claims for authorized expenses.

Flexible TimeSpan

A user-defined period into which costs can be collected. Flexible TimeSpans can be as long or as short as you like—covering multiple years or a single day. The main purpose of Flexible TimeSpans is to assist you in analyzing costs.

Flexible Work Schedule

A method of scheduling a time reporter's time that is based on a range of flex hours of start and stop times and core work hours. This method can be used for scheduling clock and elapsed time reporters (see Scheduling)

FLSA Status

A PeopleSoft Human Resources term that is used to indicate whether a job is exempt or nonexempt according to the Fair Labor Standards Act. All employees associated with a

particular job will receive that job's FLSA Status. FLSA Status is an eligibility determination factor for PeopleSoft Benefits Administration.

Forecast Attribution

A FSI (financial services industry) transformation process through which forecasted product originations are pooled and run through the cash flow engine for future periods.

Forecast Calculation Process

In PeopleSoft Demand Planning, the process by which a **Statistical Forecast** is generated for each item at each level of the view. When a **Forecast Item** is set to recalculate, the system tries several forecast calculation methods and picks the one with the least amount of error. This process also makes adjustments for promotions and filters for abnormal demand.

Forecast Definition

Forecast definitions are a set of forecasting rules that generally govern multiple forecasts distinguished by key properties such as products, customers, channels, and so forth.

Forecast Element

Each forecast within a single definition is called a Forecast Element.

Forecast Fulfillment

In PeopleSoft Demand Planning, a process used to manage forecasted demand over a period of time. The process makes it possible to divide the total forecast demand into portions so that certain portions can be met, even if the total forecast cannot be met entirely.

Forecast Item

In PeopleSoft Demand Planning, a logical item used as the basis to forecast demand. The components of a forecast item key are defined for each level in a forecast view.

Forecast Level

See Level.

Forecast Period

A period in time as defined by the calendar for which data is processed through the PeopleSoft Demand Planning model.

Forecast Start Period/Year

Determines the most recent period for which demand data is available for a forecast view. This period can also be described as the last actual demand period to have had an impact on the forecast.

Forecast View

See **View**.

Foreign Education

Education acquired outside of any state of the U.S., the District of Columbia, the Commonwealth of Puerto Rico, a Trust Territory of the Pacific Islands, or any territory or possession of the U.S.

Form 10-K

A form used for annual reports pursuant to Section 13 or 15(d) of the Securities Exchange Act of 1934 for which no other form is prescribed.

Form 10-Q

A form used for quarterly reports under Section 13 or 15(d) of the Securities Exchange Act of 1934, filed pursuant to Rules 13a-13 or Rule 15d-13. This report, which public companies are required to file quarterly with the SEC, provides unaudited financial information and other selected material.

Form 5500 Participant Count Extract

A PeopleSoft Pension Administration data extract containing data that a plan administrator needs in order to complete IRS Form 5500, used to report on the number of plan participants.

Form S-8

A form used to register securities offered by a reporting company under its employee benefits plans, including stock option plans. Also called the Registration Statement under the Securities Act of 1933.

Form W-2

A form used by employers to provide workers with a statement of wages, tips and other compensation from the previous year. This form, distributed employees by January 31 of each year, reflects state and federal taxes, social security, Medicare wages, and tips withheld.

Formula

Element which enables you to define your own formulas for use—gives further flexibility to define complex organizational needs.

Formula Plan

This compensation distribution plan type is based on a pay out rule, as the pay out rule is defined. The pay out rule can be based on a flat amount, a percentage, or a data element. Whereas a Target Plan distributes pay out based on a comparison of a performance measure against a target, in a Formula Plan the pay out is based just on the pay out rule.

French Professional Elections

French companies employing a certain number of employees must hold elections for selecting personnel representatives (Délégués du personnel), and members of the Work Council (Comité d'Enterprise).

Frozen Rate

A rate that is applied to allocate resources to activities in place of the actual, budgeted and capacity rates calculated by the Activity-Based Management (ABM) Engine.

FTE (Full Time Equivalency)

FTE is the percent of full time the employee should normally work in this job. Full time is defined by the Standard Hours specified in either the Salary Plan Table or the Default Standard Hours specified in the Installation Table.

FTP (Funds Transfer Pricing) Adjustments

Adjustments made to the PeopleSoft Funds Transfer Pricing (FTP) base rate, for such factors as geographic premiums, liquidity premiums, embedded options, or incentive programs.

FTP (Funds Transfer Pricing) Base Rate

In PeopleSoft Funds Transfer Pricing (FTP), this refers to the basic charge or credit that is applied to a ledger account, a product, or an off-balance sheet position.

FTP (Funds Transfer Pricing)

See PeopleSoft Funds Transfer Pricing.

Full-Time Equivalent

See FTE.

Function

A category of pension calculation. PeopleSoft Pension Administration divides a pension calculation into nineteen “core functions” such as Service, Final Average Earnings, and Benefit Formula.

Function Result

The calculation rules for any of PeopleSoft Pension Administration’s nineteen core functions. These rules match Definitions—the specific parameters for the function—to the Groups of employees that use that particular definition. Function Result also refers to the value produced by the rules.

Fund ID

In the financial services industry, Fund ID is a lookup code used to track investment funds associated with a financial instrument or account. Provided primarily by the financial analytic applications to track investment funds for insurance policies.

Funds Transfer Pricing (FTP)

See PeopleSoft Funds Transfer Pricing.

Fungible

This term describes a resource used for multiple activities.

Future Period

Any pay period which is not current and whose close date hasn't passed (see Time Reporting).

Future Periods

The number of periods of future forecasts maintained by the PeopleSoft Demand Planning system.

G

Gang Reporting

See Crew Reporting.

General Deduction

Any non-benefit deduction. Examples include charitable deductions, union dues, parking, garnishments, and bonds. General Deductions are calculated from the General Deduction Table; Benefit Deductions draw on one of the benefits tables.

General Ledger Distribution

The process and guidelines by which accounting information is transferred from your PeopleSoft Receivables or Deduction Management system to a general ledger system.

General Schedule (GS)

Compensation and pay plan used by the Executive Branch of the federal government.

Generation Control

Generation control elements allow you to indicate to the system whether to process an element based upon criteria you define. There are six parameters that control this function and comprise the definition of the generation control element—HR Status, HR Action/Reason, Segment Status, Frequency, Formula, and Run Types.

Generic Conversion Factor

A conversion factor that applies universally between two units of measure. The factor is used in the conversions between levels of PeopleSoft Demand Planning **Forecast Item**s and Inventory Planning **Policy Item**.

Generic Process Type

This term applies to 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," "SQR Report," and so on.

Geo RSZ Code

This code is for Belgian employers to track the geographical location for RSZ codes.

Geographic Location Code

In Canada this code is prescribed by the government and refers to the location a business is in.

Giveaway Adjustment Type

In PeopleSoft Order Management, the price break tables are set up to indicate what product the user receives as a free premium based on defined quantities or prices. The giveaway item does not have to be the same product that the customer is purchasing. For instance, you may set up a price break that indicates that a purchase of 100 widgets entitles the customer to one or more free T-shirts. The system automatically adds an order line for the free item. Giveaways cannot be applied to the total order.

Goals Matrix

In PeopleSoft Workforce Analytics, a matrix used to create calculation rules for group or employee performance goals. You can combine and standardize multiple performance goals into a single, weighted, goal score, against which actual performance is compared. A Goals Matrix can be used to in conjunction with a pay out distribution plan called a Target Plan.

Grace Period

A period that is a number of hours or minutes before or after a scheduled punch where a time reporter’s punch is accepted. For Stock Administration, the period of time an optionee has to exercise an option after termination and before the option expires.

Grade

A range of pay in a graduated scale that includes positions of different occupational groups. The work performed should be equivalent as to the level of difficulty and responsibility and the level of qualification requirements of the work. The levels are established and designated within a specific pay plan by law or regulation.

Graduate Education

Successfully completed education in a graduate program for which a bachelor's or higher degree is normally required for admission. To be creditable, such education must show evidence of progress through a set curriculum, i.e., it is part of a program leading to a master's or higher degree, and not education consisting of undergraduate and/or continuing education courses that do not lead to an advanced degree.

Grandfathered Benefit

A benefit that an employee was entitled to prior to a change in the plan and that defines the employee's new minimum level of benefits. The change might be caused by a plan merger, new legislation, or a plan amendment.

Grant

A contractual right giving an individual the option to purchase a specified number of shares of stock through an Equity Compensation Plan. Also known as an option.

Grant Agreement

The legal document issued by a company defining the number of shares granted, grant price, vesting schedule and other terms and conditions of the stock option or stock award.

Grant Date

The date the individual begins participating in a stock purchase offering. The date on which an option or other award is granted. The date the company enters into the grant agreement. The underlying stock's fair market value on this date generally derives the option price.

Grant Price

The price per share at which the stock option was granted. This is the price per share the individual must pay when exercising the option.

Gross Salary

The sum of an employee's salary and earnings defined as part of gross salary. The gross salary is used to calculate budget amounts for benefit plans defined as a percentage of an employee's salary.

Gross-up

The process used to calculate taxes and resultant gross pay from a check for an exact net amount.

Group

In PeopleSoft Billing, a specific term for a posting entity composed of one or more transactions (items, deposits, payments, transfers, matches, or write-offs).

Group

Any set of records associated under a single name or variable in order to run various calculations in PeopleSoft Business Processes. In Time and Labor, for example, employees are placed in groups for time reporting purposes, while in Administer Variable Compensation, groups identify which employees are eligible for what forms of compensation. In PeopleSoft Pension Administration, you'll use Custom Statements to define criteria for grouping employees, then by associating calculation rules (Definitions) with specific Groups, you can vary rules for different classes of employees.

Group Asset

A financial asset with no cost information. It is used to depreciate the sum of the costs of its associated group member assets.

Group Asset Depreciation

The depreciation of a group asset calculated using an average service life set by a local regulatory agency and a calculated group depreciation rate.

Group Coverage (Or Generic) Qualification Standards

Standards prescribed for groups of occupational series that have a common pattern of education, experience, and/or other requirements.

Group Member Asset

A financial asset with cost information. Cost information for all group members of a group asset is summed up to the group asset level, where depreciation is calculated.

Group Security

The ability to grant or deny access to groups. You can set up group security by Group ID or by user ID.

Group Security [Time and Labor]

The ability to grant access to employee time, by providing security through Time and Labor's groups functionality. For example, you might want your employees to only access their own records, or allow your supervisors who handle all of the time input for have access to specific groups. You can restrict the user from accessing everyone, or allow the user to be able to access only their own records, or only a specific group. This feature also provides the ability for employees to report their own time.

Group Type

An indicator of the activity that created the billing group: billing, maintenance, payment, transfers, or unposted.

H

Handicap Code

A code that identifies a type of physical or mental impairment that substantially limits one or more of an employee's major life activities.

Hazard/Disposal Code

An inventory item group sharing a disposal routine.

Headcount

The number of people represented by a given Employee Survey record in the PeopleSoft Enterprise Performance Management product line.

Health and Safety Executive (HSE)

Health and Safety reporting for your UK operations is sent to the local office of the HSE per the requirements of the RIDDOR (Reporting of Injuries, Diseases, and Dangerous Occurrences Regulations).

Health Benefits Code

An alpha/numeric code that identifies each Health Benefit plan.

Health Benefits Effective Date

Date the health benefit plan goes into effect or the effective date of cancellation.

Hierarchy

Hierarchy refers to the relationship between the levels in a dimension.

Highly Compensated Employee (HCE)

An IRS employee category applied to employees who are considered “highly compensated” according to a federally set standard. This distinction is used for the purposes of nondiscrimination tests, to determine that Section 401 and Section 129 plans do not discriminate in favor of highly compensated employees.

HIPAA

The Health Insurance Portability and Accountability Act of 1996. PeopleSoft Benefits applications enable you to comply with this act, which requires that employers provide Certificates of Group Health Plan Coverage to employees who have their health coverage terminated. This certificate lists group health coverage an employee had for the twelve month period prior to the date coverage ended as a result of termination of coverage. The HIPAA certificate will be used by subsequent health coverage carriers to evaluate pre-existing condition clauses, if applicable.

Historical Periods

In PeopleSoft Demand Planning, a component that indicates the maximum number of periods of historical demand maintained for a **Forecast Item** within a **Forecast View**. Historical periods must be a minimum of two years in order to support the development of seasonal models based on an item's demand history.

Historical Rules

An element used to set up rules that retrieve data from prior periods. Historical rules can be used in formulas and fictitious calculations.

Historical Usage Calculation Method

In PeopleSoft Inventory Planning, a method that defines the set safety stock or minimum inventory level. The usage is based on the review of historical demand over the number of effective periods. The historical demand quantity is determined by one of four methods; maximum possible usage, Lead Time, estimated daily or period use, and static values calculations.

Hold Grade/Step

Grade/step the employee was in prior to receiving a temporary promotion.

Hold Last Equivalent Increase (LEI)

Date held by an employee for this event prior to receiving a temporary promotion. Necessary in order to establish the WGI due date if returning to original grade/step.

Hold Position Description

The new position description number that is the result of a reclassification action prior to the NOA being processed.

Hold Purchase

A flag that tells the system to keep this participant in the purchase process. The hold flag is maintained at the contribution page.

Hold Within Grade Increase (WGI) Due Date

WGI due date prior to an employee receiving a temporary promotion.

Holding Period

Typically refers to the holding period required for ISO's and Qualified Section 423 Purchase Plans, to receive preferential tax treatment on a disposition of shares. See Disqualifying Disposition.

Hours Counting Service

A service calculation that uses actual or generated hours to determine the service credited to a pension plan participant.

Hours Equivalence Service

A service calculation that uses hours to determine service, but that uses a set number of hours per day, week, or other period worked rather than counting actual hours.

HR Action/Reason Category

A group of related job actions—for example, hire and rehire—treated similarly for pension purposes in PeopleSoft Pension Administration.

HRMS Warehouse

See Warehouses.

I

Ignore Plan

Complex event processing feature of PeopleSoft Benefits Administration that enables the user to designate plan types linked to a particular Event Rules/Event Classification combination as being unaffected by Benefits Administration processing.

Ignore Violations

The ability to report over capacity violations but not to score or repair them during the optimization process.

Imputed Income

Theoretical income that a company pays on behalf of an employee but the individual does not actually receive. This “theoretical income” must be added to the employee’s gross wages. In general, imputed income refers to the value of excess Group Term Life or Dependent Life coverage.

In Punch

Indicates start of a shift.

In the Money Option

When the fair market value of the stock is greater than the grant price of an option.

INAIL code

In Italy, the INAIL code is used to classify jobs according to the level of risk associated with the job and the related risk insurance required by the employer. INAIL codes are defined by the employer.

Incentive Pay Plans

In PeopleSoft Workforce Analytics, pay plans that are formula-driven based on the expected results defined at the beginning of a performance cycle. Incentive plans are designed for the individual worker, or for group levels such as teams, business units, divisions, or company-wide. Incentive plans are used for a variety of reasons; including cost control, alignment of employee and shareholder interests, and increased focus on specific performance indicators.

Incentive Plans

Pay plans that are formula-driven based on the expected results defined at the beginning of a performance cycle. Incentive plans can be designed for the individual worker or at group levels such as teams, business units, divisions or company wide.

Incentive Stock Option (ISO)

For an option to be considered an Incentive Stock Option, it must have the following characteristics:

- The option must be granted pursuant to a plan which includes the aggregate number of shares which may be issued under options and the employees (or class of employees) eligible to receive options, and which is approved by the stockholders of the granting corporation within 12 months before or after the date such plan is adopted;
- The option must be granted within 10 years from the date such plan is adopted, or the date such plan is approved by stockholders, whichever is earlier;
- The option is not exercisable after the expiration of 10 years from the date such option is granted;
- The option price is not less than the fair market value of the stock at the time such option is granted;
- The option is not transferable by such individual otherwise than by will or the laws of descent and distribution, and is exercisable, during his lifetime, only by him, and;
- The optionee, at the time the option is granted, does not own stock possessing more than 10% of the total combined voting power of all classes of stock of the employer corporation or of its parent or subsidiary corporation.

Incomplete Punch

A punch that cannot be processed (i.e. missing employee ID, invalid date or time).

Incremental Budgeting

A budgeting option during budget development that uses prior year actual or budget values as a basis and then applies a percentage that increments the base. PeopleSoft Budgeting-specific.

Incumbent

An employee currently assigned to a position.

Indirect Compensation

Typically involves non-cash types of compensation awarded to the individual in exchange for their contribution to the organization. Common types of indirect pay include health and welfare benefits (for example, medical, dental, vision, long-term disability, short-term disability, unemployment insurance), payment for time not worked (for example, holiday, vacation, sick), and employee services and perquisites (for example, club memberships, parking, holiday gifts).

Indirect Cost

A cost that is assigned by management to an activity or a cost object. An example is the cost of office space assigned to an activity.

Individual Occupational Requirements

Requirements, e.g., experience or education, for particular occupational series of positions within a series and are used in conjunction with a group coverage (generic) standard.

Individual Retirement Record (IRR)

Used by the Office of Personnel Management (OPM) as the basic record for determining the retirement benefits payable to separated federal employees and their survivors. Employees covered by the CSRS retirement plan require SF-2806. Employees covered by the FERS retirement plan require SF-3100. In addition, the SF-2806-1 and SF-3101 are used for corrections to the IRR. See also Correction to IRR.

Inherit Control Group Policies

In PeopleSoft Inventory Planning, a feature that controls whether the policy for an item is set explicitly or defaults from the associated **Policy Control Group**. A series of check boxes enable you to define which policies to inherit.

Initial COBRA Events

The event which makes an individual eligible for COBRA coverage. Typical initial COBRA events include loss of benefits eligibility due to termination, reduction in hours, retirement, and military leave, as well as divorce, death of employee, and Medicare entitlement. See COBRA and Secondary COBRA Events.

INSEE (National Institute for Statistical and Economical Studies) Codes

INSEE is an official statistics and economics organization in France. INSEE codes for your French company's organizations are used in regulatory reporting.

INSEE PCS (Classification par Catégorie Socio-Professionnelle) Code

Each PeopleSoft Human Resources French Jobcode is linked to a four-digit INSEE PCS, or social/professional classification code.

In-Service Date

In PeopleSoft Asset Management, the date upon which an asset is placed in service. In-service date is used in conjunction with an asset's prorate convention to determine Begin Depr Date.

Inservice Placement

Includes a noncompetitive action in which a position is filled with a current or former competitive service employee through promotion, reassignment, change to lower grade, transfer, reinstatement, reemployment, or restorations. Inservice placement also includes noncompetitive conversion f appointees whose Federal excepted positions are brought into the competitive service under Title 5 CFR 316.702, and Department of Defense/Nonappropriated Fund (DOD/NAF) and Coast Guard NAF employees whose positions are brought into the competitive service.

Insider

An officer, director or principal shareholder of a publicly owned company and members of his or her immediate family. This category may also include other employees of the company and people who obtain nonpublic information about the company.

Insider Trading

When a person trades a security while in possession of material non-public information in violation of a duty to withhold the information or refrain from trading. The securities law broadly prohibits fraudulent activities of any kind in connection with the offer, purchase, or sale of securities.

Instance

A row of data on the Positive Input table. Instances of positive input can be entered manually, or can be system generated. They can also be received from other applications, such as PeopleSoft Time and Labor.

Integration Level

The salary level in a defined benefit excess plan at which a higher benefit rate becomes applicable. For example, the following formula uses a $10,000 integration level: 1% of Final Average Earnings up to $10,000 plus 1.75% of Final Average Earnings over $10,000.

Integration Template

A high-level template that defines the integration between PeopleSoft Projects and your other financial applications. Each integration template you create defines a specific set of business units from your other financial applications. Each project is then assigned an integration template containing this preset integration information. You can use Integration Templates to set up joint ventures, and new transactions added to that project will reflect the business units defined in the integration template.

Intensity

The cost for each unit of the activity driver.

Interest

Some companies pay interest on the monies that are being withheld from employees' paychecks. The interest plus the employees' stock purchase contributions are used to purchase stock at the end of the purchase period.

Interest Rate Modeling

An FSI feature that allows you to model interactively interest rate scenarios for Asset Liability Management, and to run rate scenarios and analysis in real time.

Interest Rate Sensitivity Model

In the financial services industry, this support module describes in granular terms how a group of customers holding a specific type of instrument with a particular interest rate will respond to changes in interest rates in the market.

Interface Loader

An SQR delivered with PeopleSoft Asset Management that is used to transfer load lines into the PeopleSoft Asset Management loader tables.

Internal Data

Data from PeopleSoft ERP systems, or other legacy ERP systems used by your organization.

Interpolation

To calculate a value of a function, or series, between two known values.

Interunit Account

The account for each business unit to which other business units in the same corporation refer when they need to distribute amounts across business units. These accounts are used to keep the individual ledgers in balance when a single transaction affects multiple business units.

Inter-Unit Drivers

Drivers that provide a means of establishing relationships between the cost objects of one organization with the supporting activities of the organizations that share business units and models.

Interunit Transaction

A transaction that involves moving amounts from an account in one PeopleSoft General Ledger business unit to an account in another General Ledger business unit.

InterUnit Transfer

A transfer that occurs between different business units.

IntraUnit Transfer

A transfer that occurs within one business unit.

Intrinsic Rewards

A reward that is generated by the worker internally such as job satisfaction, as opposed to Extrinsic Rewards which are tangible rewards.

Inventory Adjustment

A process that enables you to change the quantity of an item in the inventory system to match the actual physical quantity found in the **Storage Location**.

Inventory Business Unit

Usually a warehouse. You establish a separate inventory business unit (IBU) for any one of the following reasons: 1) You want on hand visibility to a specific location of your business that manages inventory. 2) You want to define replenishment rules for a specific location of your business that manages inventory. 3) You maintain standard and average costs in a specific location of your business that manages inventory.

Inventory Cost Element

A cost that can be associated with inventory items and inventory transactions. Examples include freight, overhead, and transportation. Each cost element has a unique cost code.

Inventory Item

A tangible commodity that is stored in an Inventory business unit (Ship From warehouse).

Inventory Location

See **Storage Location**.

Inventory Policy

In PeopleSoft Inventory Planning, a set of rules that controls how inventory policy values are calculated for items. Inventory policy is defined at the **Policy Control Group** and stockkeeping-unit levels. The elements that make up inventory policy are order quantity, safety stock, **Reorder Point**, and minimum and maximum policies.

Inventory Transaction

An event that moves inventory into, within, or out of the inventory business unit. Examples include material transfers, inventory adjustments, and standard issues.

Inventory Transaction Group

An identifier that categorizes transactions by type for costing purposes. For example, you can group all types of interunit transfers together.

Invoice Format Identifier

An identifier for the formatting options that determine the sorting and summarization levels of invoice information.

IRC 423 (Internal Revenue Code 423)

The section of the IRC that defines a Qualified Employee Stock Purchase Plan.

IRR Fiscal Data Accumulation

This report accumulates all retirement deductions for employees, as well as any LWOP and any basic pay that was received when an employee was not covered by the CSRS or FERS retirement plans.

IRR Remarks

Special remarks that are documented on an employee's IRR. IRR Remarks can be set up ahead of time and can be system-entered text or employee-specific.

IRR Status

IRRs can be in pending or final status. Those in pending status can be updated and corrected. A final status indicates that the IRR has been processed and can't be updated or corrected except through a Correction IRR or a Supplemental IRR.

IRR Worksheet

A preliminary IRR form that enables an agency to print a pending IRR for a separated employee, review it and make corrections, if necessary. Agencies can also use the IRR Worksheet to view a current IRR for an active employee.

ISO IRS $100K Limit

The limit the IRS places on the exercisable value of Incentive Stock Options (ISOs) of $100K per calendar year based upon the fair market value at the time of grant (Section 422 of the Internal Revenue code).

ISO to NQ Grace Period

The period of time after which an Incentive Stock Option is treated as a Non-Qualified Stock Option for tax purposes upon the termination of employment according to Internal Revenue Code Sections 421 and 422. Depending on the termination reason the option is treated:

- If the termination reason is for any reason other than death or disability, and an exercise occurs more than three months from the termination date, the system withholds taxes as if the option is a non-qualified stock option.
- If the termination reason is disability, the system withholds taxes if an exercise occurs more than twelve months from the termination date.
- If the termination reason is death, the system always treats the option as an ISO.

Issue

See **Material Issue**.

Issuer

A legal entity that has the power to issue and distribute a security.

Item

See **Inventory Item Planning Item** or **Receivables Item**.

Item Content Provider

Third-party software consisting of web-based catalogs of item and price information. These systems benefit the design and purchasing of new products by accelerating item location, maximizing design reuse, and reducing acquisition costs. PeopleSoft Purchasing, Engineering, and Inventory integrate to Item Content Providers, and the information is used by many other PeopleSoft applications.

Item Rounding Rules

A set of rules determining how fractional values are rounded so that calculations result in whole numbers. Rounding rules are used in conjunction with **Quantity Precision Rules**.

Item Simulation

In PeopleSoft Demand Planning, a process that enables you to interact with the forecast in a manageable manner and perform "what-if" analysis by comparing the effects of different forecast models.

Item Type

An identifier that defines inventory items at a very high level, and may include sets of Item Families. For example, the families Computer Items and Office Furniture might be categorized by types like Outside Manufacturing, Finished Goods, and Work In Progress.

Item-Specific Conversion Factor

A conversion between the same two units of measure when the measurements have a different value for an item. For example, a conversion between packaging unit and stocking unit.

Iterative Processing

Refers to a concept on only re-calculating those payees who have had changes and need to be recalculated (if you choose to run your payroll multiple times before actually finalizing it). This concept saves you a lot of time as you only have to recalculate those payees who have had a data change or who you indicate you would like to be recalculated.

J

Java Server Handlers (JSH)

The JSH manages network connectivity, making service requests from the Jolt Repository, and translating Tuxedo buffer data into the Jolt buffer.

Java Station Listeners (JSL)

The JSL handles the work of the client connection, tracking client messages, and session handoff.

Job Code

An ID for a job as defined on the Job Code table.

Job Code Components

The pay components assigned to a job code by associating rate codes with job codes on the Default Compensation page or the Non-Base Compensation page of the Job Code table.

Job Code Cost

Evaluation of salaries for specific job codes.

Job Compensation Rate

The compensation rate of the corresponding job row.

Job Events

Actions relevant to an employee's employment—such as a hire, transfer, or termination—that can affect benefit program or plan eligibility. Used by PeopleSoft Benefits Administration. See Event Class.

Job Order Cost Accounting

A cost accounting method that attempts to develop a discrete cost for each job performed or product produced. Only the material, labor, and overhead required to complete the job are attributed to the job cost.

Joint and Survivor Payment Option

A form of pension payment in which benefits are paid for the life of the participant and a beneficiary. Should the beneficiary outlive the participant, the benefit continues (often in a reduced amount) for the life of the beneficiary.

Joint Staffing Report

In the United Kingdom governmental agencies are required submit the Joint Staffing Report. Although it is mainly designed for government sector organizations, commercial organizations may also use this SQR to provide a summary of their staffing by department, job code, gender and full/part time employment status.

Jolt

A BEA/Tuxedo companion product that runs on an application server domain and is used to listen for Web Client Java requests and transfer them to Tuxedo.

Journal Code

The second highest level of three categories for defining a financial transaction (or document), necessary when using document sequencing. Examples of journal code are domestic sales and export sales. This category is preceded by journal type and followed by document type.

Journal Generator Template

A table containing defaults to be used in journal generation. PeopleSoft Asset Management and Billing require one journal generator template for each transaction type.

Journal Line

A record storing a double-sided, balanced entry for a given journal. A single journal usually includes multiple lines. The sum of the monetary amounts for the journal lines in one journal totals zero (debits = credits).

Journal Template

A list of the characteristics of the general ledger journal entries that will be created from your PeopleSoft Receivables system.

Journal Type

The highest level of three categories for defining a financial transaction (or document), necessary when using document sequencing. Examples of journal types are sales journal and purchase journal. This category is followed by journal code, then document type within the journal code.

Journal Voucher

A PeopleSoft Payables voucher that enables you to make accounting entry modifications while keeping your PeopleSoft General Ledger and Payables systems in sync. Like the adjustment voucher, the journal voucher is linked to an existing voucher.

K

Kanban ID

A unique identifier used to track Kanban cards and replenishment requests when using PeopleSoft Flow Production.

Keep Ledgers in Sync

An option in PeopleSoft General Ledger that defines how a transaction should be posted—to all ledgers in a ledger group as opposed to only a single specified ledger.

Key

See **ChartKey**.

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.

Key Performance Indicator (KPI)

KPI is used by the PeopleSoft Performance Management analytical applications. KPIs are high-level measurements of how well an organization is doing in achieving critical success factors. A KPI defines the data value or calculation from the Data Warehouse tables upon which an assessment is determined.

KPI (Key Performance Indicator)

See Key Performance Indicator.

Knowledge, Skills, And Abilities (KSA)

Also known as Competencies, these are attributes required to perform a job and are generally demonstrated through qualifying experience, education, or training. *Knowledge* is a body of information applied directly to the performance of a function. *Skill* is an observable

competence to perform a learned psychomotor act. *Ability* is competence to perform an observable behavior or a behavior that results in an observable product.

L

Labor Costs

Actual expenditures associated with *salary* portion of time reporter expense.

Labor Dilution

A process that occurs after the Labor Distribution process in PeopleSoft Time and Labor. The labor dilution process takes the costs that the payroll system has calculated for payable time, determines an average or rate per hour, and applies the average amount evenly across all reported hours for the day.

Labor Distribution

The process of distributing payroll expense to the corresponding payable time entries generated in PeopleSoft Time and Labor.

Labor Distribution Amount

An actual labor cost associated with reported time.

Last Equivalent Increase (LEI)

Reflects the effective date of the last step received in grade or the last promotion, whichever is most current (does not include QSI). Used as the basis to establish an employee's WGI due date.

Last Physical Counting Event

The last date the inventory item was counted. This information is stored with each inventory item.

Last Purchase Date

The item's most recent purchase date in the inventory business unit.

Last Putaway Date

The item's most recent putaway date in the inventory business unit.

Last Putaway Document Number

The item's most recent putaway document identification number in the inventory business unit.

Last Receiving Date

The item's most recent receipt date in the inventory business unit.

Last Shipping Date

The item's most recent ship date in the inventory business unit.

Last Shipping Document Number

The item's most recent shipping document identification number in the inventory business unit.

Law Enforcement Officers (LEOs)

Positions within the Federal government involving law enforcement. Under FEPCA, many of these positions are entitled to additional special pays.

Lead-Time Estimated Usage

An inventory planning method for calculating historical usage of an item. The historical demand is prorated on a daily basis and then multiplied by the number of days lead time for each effective historical period. The maximum period value is then used as the safety stock or minimum stock level. This method should be used for items that have a steady demand pattern throughout each period.

Lead-Time Period Usage

An inventory planning method for calculating historical usage of an item. The purchase lead time is rounded up to a specified number of periods. The historical demand is calculated as the maximum usage during these periods and the safety stock or minimum-stock level is set to this value.

Leave

Time entitled to an employee as a benefit, such as, Sick, Vacation, STD, and LTD. This process is managed by HRMS (see Time Reporting).

Leave Accrual Processing

Processing of leave accruals is used to maintain employee leave balances. All leave benefit plans accrue leave by length of service or number of hours worked. Leave accrual processing is used to determine the employee's leave accrual award and resulting leave balance.

Leave Accruals

Hours that employees earn to use at another time, such as annual leave and sick leave.

Leave Plan

A method for earning and managing leave time.

Leave Without Pay (LWOP) Total (Cumulative)

An employee's cumulative number of hours of leave without pay (LWOP).

Ledger Group

In PeopleSoft General Ledger, a group of ledgers consisting of one primary ledger andsecondary ledgers.

Ledger Mapping

Ledger mapping is a process that enables you to relate expense data from your 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 the PeopleSoft Enterprise Warehouse (EW), Ledger Mapping enables you to map general ledger accounts to the EW Ledger table.

Ledger Template

A table containing records and fields common to all ledgers that ensures that all ledgers specified in a ledger group share the same physical layout.

Ledger Type

The unique combination of a single ledger, scenario, and fiscal year. Multiple ledger types make up a ledger type set.

Ledger Type Set

A collection of ledger types, the members of which will represent the members of your ledger type dimension.

Legend ID

A way of recording information that is displayed upon the Issuance Instruction Report. Can be used to record a notice that should appear on the back of a stock certificate indicating that the shares represented are "Restricted Securities." Can also be used to indicate how shares should be processed, as in the case of Swaps, Trades, Repurchases and SAR Exercises.

Level

A section of a tree that organizes groups of nodes.

Defines a set of **Forecast Item** with a common key structure. Each level is related in a hierarchical definition with other levels in the view. A level definition contains descriptive and control data that relates to the operation of the forecast at each level within the view.

Level

The section of a tree that organizes groups of nodes.

Level Income Payment Option

An annuity form of pension payment in which payments are increased in early years (prior to eligibility for Social Security benefits) and decreased in later years when Social Security benefits are also received. The goal is to provide a relatively constant total retirement income both before and after Social Security eligibility.

Life Profile

In PeopleSoft Demand Planning, a feature that enables you to establish product forecasts based on predefined patterns in an item's life cycle.

Lifecycle (of Reported Time)

A representation of time through the various stages of Time and Labor; includes processing of current, future, and previous period time from scheduling and time capture through Time Administration and distribution.

LIFO (Last In First Out)

Method used by companies to record Disqualifying Disposition Income. If a company uses this method they record the optionees disposition of shares by attributing the shares to the most recent exercise, purchase or release dates for which shares remain available for sale.

Line-Item Budgets

The budget amounts associated with ChartField distributions that make up an organization's budget. Line-item budgets include personnel costs as well as operating and maintenance costs. They also include revenue estimates. PeopleSoft Budgeting-specific.

Line Schedule Editor (LSE)

PeopleSoft Production Planning utility or tool that displays production tasks for multiple products on multiple resources across multiple periods of time.

Literal Mapping

In PeopleSoft Demand Planning, a mapping option for formatting data that is common to all records being imported. This enables you to set an available field value for all the loaded rows.

Load

The feature that initiates a process to automatically load information into a PeopleSoft application—for example, populating the PeopleSoft Benefits database with plan-level election information.

Load Activation

Load Activation enables you to specify exactly which part of your Data Mart to build, including security. You set up load activation on the Load Activation page.

Load Planning

The PeopleSoft Inventory feature that picks, packs, and ships orders by Load ID. Load Planning is also used to estimate shipping weight, volume, and charges.

Loader Table

Any table in PeopleSoft Asset Management used to store load lines before they are loaded into the system as open transactions. The loader tables comprise INTFC_FIN, INTFC_PHY_A, and INTFC_PHY_B.

Loan Exercise

A form of cash exercise, typically requiring a loan agreement and a promissory note.

Local Code

In PeopleSoft Demand Planning, a type of validation used for a user-defined field code. If a user-defined field is marked to require local table validation, **User-Field Code** are used to determine the list of valid values for the field.

Local Functionality

Local functionality is the set of information in PeopleSoft HRMS that is available for a specific country. You can access this information when you click on the appropriate country flag push button in the global window, or when you access it by a local country menu.

Location Accounting

An accounting method that captures and records material movement within the warehouse, providing accounting visibility based on where the inventory resides. You can designate certain **Storage Area** as raw material, WIP, or finished goods by assigning the corresponding account ChartField (account, department, product, and project ID) to the storage area. All inventory locations in a storage area use the storage area account.

Location Code

Locations enable you to indicate the different types of addresses a company has—for example, one to receive bills, another for shipping, a third for postal deliveries, and a separate street address. Each of these addresses has a different location number. Every customer role must have a primary location, which will be used throughout the system on all panels that display a customer address. The primary location—indicated by a *1*—is the address you use most often when contacting the customer, and may be different from the customer's main address.

Location Summary

A Picking Plan option that sorts the picking plan according to the highest-level sort options defined and prints the order lines and the total item quantity to pick from each **Storage Location**. Because the layout of the printed report reflects the actual positions of stock to be picked, personnel can follow a serpentine path through the warehouse, fulfilling all orders on the picking plan without revisiting locations.

Lock for Confirm

A flag on the Pay Line record that enables users to access the database 7 days a week, 24 hours a day, without affecting or interrupting payroll processing. Issues a warning message "A payroll is currently in process for this employee. This data will not be processed until the next payroll."

Log file

One way that you can monitor the build process is to review the log files that the build process automatically generates. Keep in mind that the log file is entirely separate from the script file; do not confuse the two. How much information that the log file contains is up to you. You can set up your logging so that all status (both good and bad) appears in the log, or you can specify that just the errors or warnings appear in the log. This section describes the options you can specify in regards to the Build log file.

Long-Term Variable Compensation

In PeopleSoft Workforce Analytics, a component of direct compensation that consists of long-term payments to an employee in the form of stock programs, and deferred compensation.

Lookup Codes

In the financial services industry, these are user-defined codes that enable the system to define and categorize incoming Instrument table information. They also provide a means for you to report on specific data, such as treasury position, balance type, and ledger account.

Lot Status

The status assigned to a lot. In PeopleSoft Inventory, a lot's status can be Hold, Open, Rejected, or Restricted.

Lump Sum

A tax method that determines withholding based on the Canadian Lump-Sum tax table.

Lump Sum Payment Option

A form of pension payment in which some or all of a participant's benefit is paid as a single sum.

Lump Sum Reporting

A Time and Labor process that enables you to report time in a lump sum of hours or units for a single Time Reporting Code, and quantities of time. The system uses a batch process to gather the information you enter, perform edits, and update the daily time tables. The system uses the default assignments you establish for workgroups, taskgroups, shifts and so on.

M

Maintenance Worksheet

A work space for creating write-offs, matches, or adjustments to clean up posted items.

Manage Base Pay Structure

See Base Pay Structure

Manage Compensation Planning

A PeopleSoft Workforce Rewards module that facilitates modeling and analysis of compensation costs across organization units, specific job classifications, or groups. You can focus on the impact of changes to workforce size, or on changes to fixed and variable compensation elements, and determine their effects on current and future payroll costs.

Manage Market Compensation

A PeopleSoft Workforce Rewards module you use to match your company's jobs to similar jobs found in published market compensation surveys. You then calculate a target market rate based on a weighted average from multiple surveys. This market rate is then used to assess your company's gap to market and to perform cost impact analysis.

Manage Retention Planning

A PeopleSoft Workforce Rewards module that enables organizations to analyze the factors that lead to employee turnover, and how retention of key employees affects business performance and goals.

Manual Checks

Any checks calculated and prepared outside of the PeopleSoft Payroll system that you must enter into the system manually.

Manual Count

A PeopleSoft Inventory procedure in which you enter the actual count data and then create the counting event with its header, item records, and count quantities.

Manual Events

Events that are inserted by the user manually through the BAS Activity table. Events are actions that occur, which potentially change employee benefit coverage eligibility—see Event Class for more information. Used by PeopleSoft Benefits Administration.

Manufacturing Cost Element

A particular category of an item's cost. For example, when you produce a subassembly that has a cost of $100, the cost can be broken down further into material costs, labor costs, and overhead costs.

Manufacturing Execution Systems (MES)

Third-party system that enables detailed planning and execution of production activities from production order release to completing finished goods. PeopleSoft Manufacturing integrates to MES.

Manufacturing Task

Any job that can be performed within your manufacturing facility. A manufacturing task is associated with the work center in which the task is completed.

Map File

A file that defines the relationship between fields in a third-party system and PeopleSoft Demand Planning tables.

Mapper Type

This defines whether you are mapping actual or budgeted general ledger line items to resource ID within PeopleSoft Enterprise Performance Management.

Marginal Tax Rate

The tax rate that applies to the next dollar of income generated.

Market Compensation

A compensation review process in which you match your company’s jobs to similar jobs found in published market compensation surveys, for the purpose of establishing new target market rates. Also referred to as Market Based Pricing or Market Analysis.

Market Capitalization

The value of a corporation as determined by the fair market value of its issued and outstanding common stock. It is calculated by multiplying the number of outstanding shares by the current fair market value of a share. Analysts look at market capitalization in relation to book, or accounting, value for an indication of how investor’s value a company’s future prospects.

Market Rate

Compensation rates, usually for regular base compensation or total cash compensation, found in published salary surveys. You use the Market Compensation module in PeopleSoft Workforce Rewards to age and weight this data, to create market rates you can compare against your organization's current pay rates.

Mark-to-Market (MTM) Model

In the financial services industry, the reevaluation of a portfolio's position at current market levels.

Market Variance

A comparison of the difference between an individual's, or group's, actual compensation, and available market compensation data for a comparable population in industry. Market compensation data is usually tied to job codes, and comparisons are usually made between similar jobs. Although the variance to market can be evaluated for any of the compensation components in the Compensation tree hierarchy (such as Total, Direct, or Base), market compensation data is most typically available for, and used in evaluating Base Pay (Base Salary). The main point of reviewing the market variance is to evaluate how well your workforce is paid in comparison to both prevailing compensation in industry, and your own organization's compensation strategy.

Mass Adjustment

A process of applying an amount or percentage change to one or many line item budgets at once. PeopleSoft Budgeting-specific.

Mass Cancellation of Requisitions and Purchase Orders

A utility that allows you to select and cancel groups of requisitions and purchase orders. You can use this utility during the year as well as at year-end in preparation for closing. The utility enables you to specify ChartField criteria for selecting documents for cancellation. For example, you can select all requisitions or purchase orders for a particular fund and organization, which have a remaining balance. Then you may select a subset of those records to approve for cancellation.

Mass Change

A user-configurable entity that defines the movement of data between the tables that store your business information. Mass Changes enable you to define the criteria by which you move or replace data in your tables. Based on the configuration of your system, Mass Change dynamically builds data access and gives you complete control over your system processing.

Mass Change Template

The foundation for defining mass changes. Mass change templates enable you to control which fields will be available for the operator to specify when defining a mass change, and whether those fields will be used as selection criteria or defaults.

Mass Change Type

The building blocks used in defining mass change templates. Mass change types specify which records the resulting mass change will select From the database, alter, and subsequently write back To the database. They also set up system field defaults that run behind the scenes to ensure that this mass change is processed correctly.

Mass Validate Metadata Utility

A PeopleSoft Enterprise Warehouse utility that enables you to validate, but not compile, Metadata objects. Mass Validate certifies all "as of dates" created for Filters, Constraints and DataSets for the specified run date. This utility helps ensure that your Metadata is valid at run time and increases your chance of a successful engine run.

Match

A process in PeopleSoft Workforce Planning, by which the system compares the roles, competencies, and accomplishments in the current competency inventory, with the requirements of a given competency strategy.

Matched Punches

A period between two consecutive punches during which some activity happens measured intervals.

Match-Funding

In the financial services industry, Match Funding refers to funding an asset with a like (term to maturity) liability. This helps an organization apply the appropriate funds transfer price. Although the actual asset might be funded with shorter-term liabilities, it does provide a better measure of financial performance for that asset, such as Risk Adjusted Return on Capital.

Material Costing

An inventory accounting method that assigns a cost to items in inventory. These costs can be assigned equally across all items or tracked individually for each item.

Material Issue

An event that triggers stock fulfillment requests for items in inventory.

Material News

Company news that could be expected to affect the value of a company's securities or influence investors' decisions. Material news includes information regarding corporate events of an unusual and non-recurring nature, news of tender offers, unusually good or bad earnings reports, and a stock split or stock dividend.

Material Release

A PeopleSoft Manufacturing process that—after material has been picked—decrements on hand inventory balances for the inventory storage areas and increments inventory to the WIP

locations defined by the routing or production area. The process also changes the production ID's or production schedule's status from Released to In Process.

MAX Method

See Maximum Method Policy.

Maximum Compensation Hours

The greatest number of hours to be paid for a specified TRC (see Time Reporting).

Maximum Lead-Time Usage

In PeopleSoft Inventory Planning, a policy control value that sets the safety stock level to the maximum quantity required during the lead time. This method is normally used when the demand for an item is low or intermittent but sufficient stock must always be available.

Maximum Method Policy

In PeopleSoft Inventory Planning, a policy that controls the way in which the system determines a reasonable high limit for the maximum inventory level of an item. The system provides warning messages when the inventory level exceeds the maximum level.

Maximum Taxable Wage Base

An annual earnings threshold used for Social Security purposes. Pension plans sometimes provide different levels of pension benefits for earnings above and below the Maximum Taxable Wage Base.

Measure

A measure represents the amounts brought into a cube—the numerical data.

In data warehousing, a Measure is a field type used interchangeably with fact. Measures are types of amounts. Any numeric field you want to apply a Data Manager rule against should be a measure.

Measure ID

In the Define Market Compensation module of PeopleSoft Workforce Rewards, a Measure ID is the identification code for a measure. For market compensation surveys, the Measure ID describes the percentile for each type of pay, as well as the regression statistic type. In PeopleSoft Workforce Analytics, for Benchmark Surveys, the Measure ID describes the type of benchmark.

Measure Value

In PeopleSoft Workforce Rewards, Measure Value is the calculated market rate value from market compensation surveys for a given percentile of a market rate, and for a given scenario and job code. This is the annual rate you compare against the compensation paid for similar jobs in your company. The Measure value can also be the regression statistic value used for

Regressing Market rates. In PeopleSoft Workforce Analytics, for Benchmark Surveys, the Measure Value is the delivered Benchmark Ratio.

Member

A member is the OLAP equivalent of a node or detail value on a PeopleSoft tree. A member is a single item within a dimension, such as a single product name, department ID, or part number. Member names must be unique, even across dimensions. Cube Manager uses the term Dimension Field Mapping to identify members, dimension parents, and label mappings.

Merchant

In PeopleSoft eStore and Mobile Order Management, a level of online (web or wireless device) display and order processing controls set by the seller. See also Merchant Variant.

Merchant Variant

Associated with a user ID, a subordinate level of merchant controls that enable customers to access different variations of the same PeopleSoft eStore website. In PeopleSoft Mobile Order Management, variations are primarily used to differentiate order processing options and fulfillment methods or locations.

Merit Matrix

In PeopleSoft Workforce Analytics, a matrix used to define the amount rules for base compensation increases for your workers. A Merit Matrix provides the salary increase parameters for each review rating in a rating scale. The salary increase parameters are expressed in terms of a percentage. The percentage increase amounts are usually structured to express the company's pay strategy relative to employee performance, and the employee's degree of range penetration in their salary range.

Merit Matrix Increase

In PeopleSoft Workforce Analytics, an increase to an employee's base pay awarded based upon a Merit Matrix.

Message definition

The object definition specified in Application Designer which contains message information for PeopleSoft's Application Messaging system.

Metadata

Information about data. Metadata is the information a database or application stores to describe your business data. At its simplest, metadata defines the structure of a data field—its data type and size, for example. Metadata can also describe more complex data relationships, such as the rollup structure for a chart of accounts. Reporting and analysis tools should be able to use this metadata to let users access data just as they would from within the application, without having to understand how it is stored.

For Enterprise Performance Management, metadata is used to describe the data stored in the PeopleSoft Enterprise Warehouse. There are different types of metadata, for example, TableMaps, DataMaps, and constraints. You typically define these when you set up the warehouse; however, Metadata (particularly constraints and DataMaps) is used to develop business rules that manageaspects of the dimensional models. Metadata enables technical users to define relationships between warehouse tables and enables business users to easily identify the data that interests them without having to know the database structure.

Meta-SQL

Meta-SQL: Meta-SQL constructs expand into a platform-specific SQL substrings. They are used in functions that pass SQL strings, such as in SQL objects, the SQLExec function, Application Engine programs, and so on.

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.

Method

A method can only be executed from an object, using dot notation. You have to instantiate the object first, before you can use the method.

For Enterprise Relationship Management, a method is the algorithm or formula that defines how the budget amount for a line-item budget is calculated or how it is derived if a calculation is not necessary. Types of methods include amount per FTE, itemization, annual percent growth rate based on a historical figure, and number of units multiplied by cost per unit. PeopleSoft Budget Planning-specific.

Method Amount

The amount for a line item budget resulting from the application of a method. It represents the budget amount after the method is applied to a line-item budget but before any adjustments or allocations are applied. PeopleSoft Budgeting-specific.

Method Base

The defined value to which a method is applied, if applicable. Not all methods require a base. PeopleSoft Budgeting-specific.

Method Driver

The factor used in a method's algorithm. For the method, amount per FTE, FTE is considered the driver. PeopleSoft Budgeting-specific.

Method of Payment

In PeopleSoft Grants, designates whether a payment is to be through a cost invoice or a Letter of Credit.

Method Parameter

A defined and derived value within a method, which drives an expense or revenue calculation. For the method, Amount per FTE, the number of FTEs is considered the driver parameter. PeopleSoft Budgeting-specific.

Metric

A metric is a calculation of facts. A metric is usually a number, but can be anything you want to measure.

Metric Object Security

Metric Object Security determines whether an individual can see a metric object in a Data Mart.

Midpoint (Pay Range Midpoint)

In PeopleSoft Workforce Analytics, the middle value in a pay range, halfway between the minimum and the maximum, calculated as (Minimum + Maximum)/2.

Midpoint Progression

In PeopleSoft Workforce Analytics, the percentage difference from one grade midpoint to the next higher-grade midpoint, calculated as (Midpoint2 – Midpoint1)/Midpoint1.

MIN Method

See Minimum Method Policy.

Minimum Benefit

See Grandfathered Benefit.

Minimum Compensation Hours

The lowest number of hours to be paid for a specified TRC (see Time Reporting).

Minimum Method Policy

In PeopleSoft Inventory Planning, a policy that controls the way in which the system determines a reasonable low limit for the minimum inventory level of an item. The system provides warning messages when the inventory level drops below the minimum level.

Missed Punch

A punch that is not entered at the scheduled time (see Time Reporting).

mkvdk

Verity's command-line tool used to index a collection, insert new documents, perform simple maintenance tasks like purge and delete a collection, and control indexing behavior/performance.

MLS

Multilingual support.

Modal transfer

Modal transfers allow you to transfer an operator from one component to another component (the modal component) modally; that is, requiring the operator to OK or Cancel the modal component before returning to the originating component.

Modal transfers give you some control over the order in which the operator fills in pages. They are useful for finite tasks related to the main transaction. They are particularly useful in cases where data in the originating component can be derived from data entered by the operator into the modal component.

Model Equivalency Factors

In PeopleSoft Demand Planning, factors that adjust model errors to allow a fair comparison. During the Model Reset process, the errors associated with each of the models are multiplied by their associated factors. The factored errors are then compared to select the model with minimum errors.

Models

In the PeopleSoft Enterprise Warehouse, Models enable replication of an organization's business processes for analysis of cost flow through customers, departments, and channels.

Model Recalculation

In PeopleSoft Business Planning, users may checkout slices of the entire model for their appropriate role. This requires the entire model to be periodically recalculated to incorporate the users changes for dependencies in other areas of the model.

Morphing

Morphing is a technique of automatically transforming the look and feel of an interface based on the needs of an active object. The Application Designer toolbar and menus dynamically transform based upon the type of object definition that is active.

Mortality Table

A table showing rates of death by age. Mortality tables are part of a pension plan's actuarial assumptions.

Moving Average

In PeopleSoft Demand Planning, a model that averages a selected number of the most recent demand periods and creates a forecast of demand for the next and subsequent periods.

Multibook

A functionality supporting the requirement of a company to carry one set of books in their local currency (functional currency) and another set of books in the currency of their parent company (reporting currency). In PeopleSoft General Ledger, multibook functionality is multiple ledgers having multiple-base currencies defined for a business unit, and the option to post a single transaction to all base currencies (all ledgers) or to only one of those base currencies (ledgers). Also commonly known as dual-book.

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.

Multidimensional Analysis

A type of analysis that enables you to look at data from many different dimensions, or attributes. You identify the dimensions of the data, then combine the dimensions in various ways. For example, you might identify five dimensions of your sales data: sales, region, channel, product line, and time. Once you've identified the dimensions, you can "slice and dice" the data based on combinations of these dimensions, such as sales in the Western region for the last quarter.

Multidimensional Database (MDDB)

A database that stores data for multidimensional analysis in a proprietary multidimensional format. Users access MDDBs exclusively for reporting and analysis, never transaction processing, so they are optimized for retrieval speed.

For Enterprise Performance Management, a Multidimensional Database stores data for multidimensional analysis in a proprietary multidimensional format. These databases are used exclusively for reporting and analysis, and never transaction processing, so they are optimized for retrieval speed.

Multiple Jobs

Multiple jobs allow you to hire an employee into more than one concurrent job and have them processed through Payroll, Benefits, and Pension. In order to enable this feature, the Multiple Jobs check box must be selected in the PeopleTools Options page.

Multiple-table dynamic tree

The user drills down through a hierarchy of parent and child records.

Multivariate Forecasting Techniques

In Enterprise Planning and Simulation, this is a forecasting method that uses both the recorded history for the target value and the history and forecasts for other variables (causal factors) to infer, not only a forecast for the target value, but also a functional relationship between the causal factors and the target value.

N

National Association of Securities Dealers, Inc. (NASD)

Self-regulatory organization of the securities industry responsible for the regulation of The NASDAQ Stock Market and the over-the-counter markets. The NASD operates under the authority granted it by the 1938 Maloney Act Amendment to the Securities Exchange Act of 1934.

National ID Number

Different countries track some form of National ID for payroll, identification or benefits purposes. For example, German workers are assigned a Social Insurance Number, UK workers have a National Insurance Code, and US laborers have a Social Security Number. Each of these different types of National IDs has unique formatting requirements associated with them as well.

Nature Of Action (NOA) Code

Indicates the type of personnel action being processed.

Nature Of Action Description

Describes the NOA code.

Nature Of Action Effective Date

The date the personnel action is effective.

Negative Amortization

Occurs when a loan payment does not cover the interest due on the loan payment, resulting in an increase of the principal amount.

Net-To-Zero Adjustment

A prior period adjustment where no compensation affecting fields on the pre-existing (original) record are changed by the adjustment.

New Hire Report

In the United States the Personal Responsibility and Work Opportunity Act of 1996 (the so-called Deadbeat Dads law) requires employers to report new hires to specified agencies within a pre-determined number of days from the hire date.

Next Level Item

In PeopleSoft Demand Planning, the **Forecast Item** at the next level that contains the current item as a child. This is the key of the group item at the next level up and is always within the same view.

Next Year

PeopleSoft Benefits term referring to the next open enrollment processing year.

NIC (Numéro Interne de Classement) Code

In France NIC numbers identify the entities inside the same enterprise, and represent an Internal Filing Number.

No Control

A target control that allows the user to submit a budget even if it is not within the planning target and the tolerance levels. The system tracks the budget against the defined planning targets but does not generate any warnings or validations. Users can still compare their planning targets against their budget amounts on the Planning Targets page in Line Item Budgeting.

Node

An individual item on a tree. Nodes summarize detail values or other nodes, and may or may not roll up into other nodes or levels.

Node

A node is a name that you can use to refer to some source of HTML content. In more practical terms, a node is a URI string that defines the database and server to be used when the portal servlet attempts to retrieve content, proxy addresses, and assemble pages.

Non-Base Pay

A pay component not included in the job comp rate calculation. It is used by payroll only in the paysheet calculation. For example, non-base pay can be set up for additional work, holiday pay, bonuses, and so on.

Non-Benchmark Jobs

See Benchmark Jobs.

Noncompetitive Action

An appointment or placement in a position in the competitive service that is not made by selection from an open competitive examination, and that is usually based on current or prior Federal service. A noncompetitive action includes:

- All of the types of actions described under inservice placement, above
- Appointments of non-Federal employees whose public or private enterprise positions brought into the competitive service under Title 5 CFR 316.701; and
- Appointments and conversions to career and career-conditional employment made under special authorities covered in 5 CFR 315, Subpart F.

Nondiscrimination Tests (NDT Tests)

Tests used to help employers ensure that their organization's 401(k), 401(m), and Section 129 dependent care reimbursement plans do not discriminate in favor of highly compensated employees. See Highly Compensated Employees.

Non-Employee

Those workforce resources hired to perform a specific job and/or hired for a specific period of time. Although non-employee time will be entered into Time and Labor for the purposes of managing their Task time, non-employee earnings will not be updated to Payroll and they will not be paid through the Payroll system.

Non-HR Employee [Time and Labor]

An individual employed by the corporation who is administered outside of the PeopleSoft Human Resources system.

Non-Job Event

Actions which result in changes to an employee's personal or demographic information that also affect benefit program and plan eligibility—such as an a state or postal code change, a family status change like a divorce, or a birthdate change. Used by PeopleSoft Benefits Administration. See Event Class.

Non-Productive Time

Any employee scheduled work time spent on tasks (or non-tasks) other than those which the employee was hired to perform. This could include time spent in training, time spent in meetings, travel time, and time spent reporting time.

Non-Qualified Dependent

Dependents such as domestic partners, their children, and other people who do not meet the definition of qualified dependents as presented in IRS Section 152. PeopleSoft Benefits applications enable the creation of benefit programs that offer health and life coverage to non-qualified dependents.

Nonqualified Plan

A plan that doesn't conform to ERISA rules. Employers cannot take a tax deduction for contributions to a nonqualified plan; instead, plan benefits are generally paid directly from the employer's assets.

Nonqualified Stock Option (NQ)

Any option that does not satisfy the conditions of a statutory stock option under the Internal Revenue Code and therefore does not qualify for preferential tax treatment. Generally, companies can design nonqualified options in almost any way they like. Features are:

- The grant price may be less than fair market value (with some exceptions under state law).
- Grants are not limited to employee of the company or subsidiary.
- No taxable income is recognized at the time of grant.
- Options can be granted to anyone (Employees, Consultants and Board of Directors).
- Difference between the fair market value on the date of exercise and the grant price is treated as compensation income.
- In the U.S., withholding tax obligation arises at the time of exercise.
- Company receives a tax deduction equal to the compensation income recognized.

Nontaxable Benefits

Any employer contributions that are not subject to Federal Withholding Tax, such as an employer's portion of a 401(k) plan.

Normal Form of Payment

The payment form associated with the amount calculated by the benefit formula. Pension Administration uses it as a basis for converting to optional forms of payment.

Normal Hours

The hours an employee is normally expected to be at work for any given workweek.

Normal Line Of Promotion (Career Ladder)

The pattern of upward movement from one grade to another for a position or group of positions in an organization.

Normal Retirement Date (NRD)

The date on which an employee is eligible to retire and begin receiving pension benefits. Eligibility for normal retirement is typically based on age only.

Normalized database

A normalized table adheres to certain standards designed to improve the productivity of the database user. Normalization makes the database much more flexible, allowing data to be combined in many different ways.

The standards for a normalized database are called forms, such as first normal form, second normal form, and so on.

Normalized Loss

In the financial services industry, Normalized Loss is the expected loss on a loan and is netted out of the profit and loss statement for management accounting or profitability measurement purposes. Similar to the bank's loan loss reserve, it enables the institution to analyze and account for expected losses on a more detailed level, by financial product.

Northern Ireland Report

In the United Kingdom the Fair Employment (Northern Ireland) Act of 1989 requires private sector employers with more than 10 employees to submit the Northern Ireland report to the Fair Employment Commission annually. The report indicates the religious composition (referred to as Community Background—Catholic, Protestant, Other) of the workforce, job applicants and appointees.

Not To Exceed (NTE) Date

Types are as follows:

- Appointment NTE Date: Indicates the length of time a person may serve in a position.
- Classification Temporary NTE Date: Established temporary date that is used for a temporary classification of a unique position.
- Hospitalization coverage.
- LWOP NTE Date: NTE date is the last day the employee is in leave without pay status. The employee is scheduled to return to duty the next workday.
- Position NTE Date: Indicates the length of time a position is available for use.
- Promotion NTE Date: Specific NTE Date: Specific time for an increase in grade on a temporary basis.
- Suspension NTE Date: Specific time an employee is to be on suspension. No salary is paid for the period.

nPlosion

A PS/ nVision feature that enables you to expand rows and columns in your spreadsheet to underlying details, as in drilldown.

Numeric constant

Numeric constants are any decimal number used in PeopleCode.

O

Object-Based Modeling

Object-Based modeling technology enables you to create parent and child models. In the PeopleSoft Enterprise Warehouse, you set up such models using the Scenario Manager.

Object reference

An object reference is one that uses the current object. For example, in the case of a component, pages within the component are related objects. The menus that use the component are its object references.

Occupant Of Position/Vice

Indicates new position or former occupant of a position.

Occupational Series Code

Designates a grouping of positions similar in work and qualification requirements. They are designated by a title and four digit number (e.g., the Accounting Series, GS-0510).

Off Date

A specific date that is defined as an off day (see Scheduling).

Off Day

A 24-hour period rounded by daybreaker with no associated shifts (see Scheduling).

Off Day Type

A classification of off days (i.e. holiday, plant shutdown) (see Scheduling).

Off-Cycle Processing

The process of calculating and creating a paycheck for one or more employees aside from the normally-scheduled *(on-cycle*) payroll run for their pay group. You typically use off-cycle payroll processing for employees who are being terminated, new hires who weren't entered into the system in time for the last on-cycle payroll run, and employees who received an incorrect paycheck during a normal on-cycle payroll.

Offer Period

This is the period of time in which an employee’s ESPP share price is determined.

Officer

An insider who sits on the Board of Directors and who is also an employee of the corporation. Examples include CEO, CIO, CTO, CFO, COO, Corporate Secretary, and Treasurer.

Official Forwarding Address

An employee's mailing address following separation.

Official Languages Act (OLA)

Canadian federal institutions are required to report on the official languages used in their departments, in accordance with the Official Languages Act (OLA).

Official Personnel Folder (OPF)

The repository of a Federal employee's official documents related to Personnel history.

Official Personnel Folder (OPF) Address

Indicates the address where the Official Personnel Folder is maintained.

Off-Invoice Discount (OI)

A per unit discount deducted from the customer invoice and given by a manufacturer for promotional activities. Off-invoice discounts can originate from a National Allowance or Customer Promotion, and are passed to PeopleSoft Order Management so the discounts are applied correctly during order entry.

Offset Plan

A pension plan where the benefit formula includes an offset of a portion of the participant's Social Security benefits.

OLAP

Online Analytical Processing. OLAP is the multidimensional analysis of application data, performed interactively. The acronym contrasts with OLTP (Online Transaction Processing), which is what most production business application systems do.

OLTP (Online Transaction Processing)

OLTP refers to the applications that perform the business transactions that keep your company running, such as processing invoices or enrolling employees in benefits programs.

Ontario Employment Equity Commission (OEEC)

The OEEC requires employers in Ontario to complete workforce surveys.

Open Enrollment

The scheduled annual re-enrollment of plan participants into appropriate benefit programs and, within those programs, benefit options.

Open Price

The price at which a security starts a trading day.

Open Season

A time period during which Federal employees are open to re-enroll in a specific benefit plan and option. Open Seasons can be scheduled at varying times throughout the year and multiple Open Seasons can occur concurrently with each other. For FEHB processing, it is generally the time period from mid-November through mid-December. For Thrift Savings Plan (TSP) processing, these are semi-annual and are generally held from May 15 - July 31 and November 15 - January 31. Open seasons for FEGLI are infrequent and special notification from the OPM would be issued to all Federal employees should they occur.

Open Transaction

A transaction that has not yet been processed in PeopleSoft Asset Management.

Operation

In PeopleSoft Manufacturing, a job or task performed in a specified amount of time, done in one work center, and using one or more resources.

Operational Data Store (ODS)

A staging area in PeopleSoft Enterprise Warehouse for source application data and pre-processed data for tables optimized for reporting.

OPF Code

Indicates where the OPF is maintained.

OPM

Office of Personnel Management.

Optimization

In PeopleSoft Demand Planning, the process of evaluating and improving forecast model parameters.

Optimize

The process of creating a new PeopleSoft Planning schedule by repairing the violated constraints in a schedule automatically. The Optimizer can be prioritized for meeting due dates, for minimizing overtime costs, and so on.

Option

A contractual right that gives the individual the option to purchase a specified number of shares of stock through an Equity Compensation Plan. Also known as a grant. Regulatory agencies also refer to an option as the right to purchase stock in an employee stock purchase plan. These options are considered granted on the offering begin date.

Option Adjusted Cost (OAC)

In the financial services industry, the difference in the average expected return between an instrument without embedded options that are otherwise identical to the fully loaded instrument and the instrument fully loaded with embedded options.

Option Adjusted Spread (OAS)

In the financial services industry, the average return expected for an instrument, over the short-term risk-free rate, for all projected interest rate paths generated using Monte Carlo simulation.

Option Types

Types of stock options. PeopleSoft Stock Administration supports the following stock option types. Incentive Stock Options (ISO), Nonqualified Stock Options (NQ), Tandem Incentive Stock Options/Stock Appreciation Right (ISO/SAR), Tandem Nonqualified Stock Option/Stock Appreciation Right (NQ/SAR), Restricted Stock Award (RSA).

Optional Forms of Payment

Any alternative forms of payment available to a participant retiring under a pension plan. These can include: annuity options paid over the participant's (and possibly a beneficiary's) lifetime; certain term options paid over a specified number of years; and lump sum options paid out in a single payment.

Options Outstanding

The total number of option shares held by optionees. It is the number of Grants less the number of Exercises, Cancellations, and Expirations.

Order Group

Order groups link order terms that default into sales orders and quotes when you select an order group code.

Order Line Number

The line associated with an order identification number. The order line identifies an item and the requested quantity.

Order Quantity Policy

In PeopleSoft Inventory Planning, a policy that determines how replenishment order quantities are calculated for an item. For example, you can use a static number, provide upper and lower limits, or use an economic order quantity calculated by the system.

Ordinary Income Tax

An individual's tax on earnings from wages, tips, and all other sources except capital gains. Includes option profits upon exercise of non-statutory options.

Origin ID

A code that identifies the location of a payment deposit in PeopleSoft Billing. Origin ID also distinguishes the method of the payment's entry: online, external, or lock box interface. Billing origin ID identifies the remit to origin for billing. This function is mainly used for specifying where the customer should send payment.

Original Option

A stock option that is eligible for repricing. This option has a grant price greater than the current FMV.

OSHA 200 Occupational Injury and Illness Recordkeeping Log

In the United States this record-keeping logbook meets reporting requirements for reporting occupational injuries and illnesses to the Occupational Safety and Health Administration (OSHA). It lists the case numbers and details of each injury and illness that occurred during a calendar year.

Out Punch

Indicates the end of a shift.

Out-of-the-Money

A term used to describe an employee stock option when the current market price is below the option grant price. When an option is out-of-the-money, it costs more to exercise than the underlying stock is worth. Such options are also described as being "underwater."

Output Result Tables

Refer to the database tables that are populated with information at the end of each pay calculation.

Output VAT

VAT collected on sales or outputs.

Outside Scope of VAT

A transaction determined as not subject to VAT. No VAT code is associated with this type of transaction. The transaction is still logged in the VAT transaction table, but no tax is applied.

Outside The Register Appointment

An appointment in the competitive service made under an agency's applicant supply system because either there is not a sufficient number of eligibles on the appropriate register or no competitor inventory exists. Agencies are also authorized to make temporary limited appointments outside the register at grades GS-12 and below.

Outstanding Option

A stock option that still has unexercised (vested or unvested) uncancelled or unexpired shares. Options with a "pending" status are not included. Only options with a status of 'active' or 'suspended' are considered outstanding.

Overlapping Promotions

Multiple customer promotions related to the same customer, and the same product, at the same time.

Override Rate

Cost per hour or unit reported with time used to replace the time reporter's default rate. (see Time Reporting)

Override text

Text not derived from field descriptions.

P

Package level

The top level of organization is the package level. The package is the entire transaction set file, addressed to your company much as a mail package would be.

Page

A page defined in Application Designer as part of a PeopleSoft Internet Architecture application.

Page Assembly

Page assembly is one of the functions of the portal servlet. Page assembly involves intercepting the user's content request, retrieving the content, and properly formatting it using a pre-defined portal template. To complete the page assembly process, the portal servlet merges content from any HTML documents that it retrieves along with the defined template

HTML. The assembled page is then sent back to the user's web browser as a single HTML document.

Page buffer

Consists of rows of buffer fields that hold data for the various records associated with page controls, including the primary records, related display records, derived/work records, and translate table records.

Pagelet

A page designed to appear on a customized homepage. A pagelet is smaller than the typical page dimensions in many PeopleSoft applications. It can be based on either a page designed in Application Designer or on an iScript.

Paired Punches

Two punches for the same employee in chronological order that exists for the purpose of determining the duration between the punches.

Par Value

The nominal or face value of a security. It establishes a price floor below which shares may not be issued. With common stock, the company issuing the stock sets par value. Par value has no relation to fair market value. Some companies issue no par value stock.

Parallel Processing

In the PeopleSoft Enterprise Warehouse, parallel processing is a system function that "locks in" the information you use for processing. This enables the system to run identical or similar processes at the same time without impacting your results. Running concurrent processes greatly reduces the amount of time it takes to run within the system.

Parent Budget

In commitment control, you can build a hierarchy between different budgets, such as summary and detail budgets. Specifying a relationship of parent and child between a summary and a detail budget for purposes of budget inquiries enables you to retrieve information about either budget through the other.

Parent/Child Models

Object based modeling technology enables you to create parent and child models. In the PeopleSoft Enterprise Warehouse, you set up such models using the Scenario Manager.

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.

Parent Task

A higher-level Planning task in a schedule's hierarchy that drills down into subtasks. Its start time is the start time of its earliest subtask, and its end time is the end time of its latest subtask. Planning tasks are distinct from Manufacturing tasks.

Partial Pay

The pay processed whenever a job record has an effective date in the middle of a pay period. Typically, this happens whenever you hire, terminate, transfer, or change the rate of pay for an employee mid-period.

Participants

Individuals who elect to participate in the stock purchase plan.

Participation

The PeopleSoft Pension Administration function that determines whether an eligible employee has met the plan's rules for joining the plan. Generally, these rules are based on age and service criteria.

Participation ID

In the financial services industry, this is a lookup code used by the financial analytic applications to identify the participants (syndicators) involved in, or responsible for, a financial instrument or group of instruments.

Passive Control

A target control allowing the user to submit a budget even if it is not within the planning target and tolerance rules. The system responds by sending an email to the user of the next budget center level indicating that the budget exceeded planning target tolerance levels.

Passive Events

Events that are initiated by a change that has taken place over time, rather than by a direct data entry action. Events are actions that potentially change benefit coverage eligibility. Examples of passive events include an employee’s reaching the age of retirement. See Event Class for more information.

Pattern Reporting

A Time and Labor process that enables you to report a start and stop date, a pattern of one or several time reporting codes, associated hours, amounts, or units and task information once for an employee. The system transforms the information into instances of daily time for each scheduled employee work day based on the employee’s schedule.

Pay

Types of "pay" are as follows:

- Basic Pay: generally, the total amount of pay received during any one calendar year at the rate fixed by law or administrative action for the position held by the employee or judicial official prior to any deductions and not including any special payments or premium pay.
- Gross Pay: total compensation earned by an employee, annuitant, or survivor of a judicial official prior to any deductions. Includes basic pay plus locality pay; availability pay (if any) for LEOs; special payments (if any); an annuity (if any); plus awards (if any).
- Premium Pay: pay provided to an employee as a regular addition to basic pay (e.g., administratively uncontrollable overtime (AUO), availability pay, overtime, night differential, holiday pay, etc.).

Pay Basis

A code indicating the principal condition in terms of time, procedures or criteria, that serves as a basis for computing an employee's pay.

Pay Calculation

Formula that calculates an employee's gross to net.

Pay Calendar

Payroll processing cycle for a given pay group.

Pay Components.

Rows in the compensation record. They build the compensation packages in the compensation record.

Pay Confirmation

Process in which the system updates all to-date totals on the database for earnings, deductions, and taxes for pay groups assigned to a given Pay Run ID.

Pay Entity

A pay entity is the organization responsible for making payments to payees. You can also use a pay entity to define the type of currency to be used when processing calculations. The pay entity is a legal definition of an organization from a payroll perspective. In many cases, an organization and a pay entity are the same. However, PeopleSoft Global Payroll does not define a relationship between an organization and a pay entity.

Pay Frequency

Defines how often employees in a pay group are paid—weekly, biweekly, monthly, and so on.

Pay Group

A set of employees grouped together for payroll processing. It's a way of "bundling" payees for more efficient processing. A pay group is made of payees that the system processes at the same time during a pay run.

Pay Period

The established time segments for which employees in a pay group are paid. Pay Periods are defined by their beginning and ending dates.

Pay Plan

A code that denotes the pay schedule under which an employee is paid , e.g., JS, UG, UJ, etc.

Pay Slip

Either an actual check or an advice notice of a direct deposit. You build these to match your organization's needs. A pay slip is the details of a payment you've made.

Pay Structure

In PeopleSoft Workforce Analytics, Pay Structure consists of a series of pay ranges or grades, each with a minimum and maximum. You develop pay structures to support and reinforce your company's pay strategy (for example, to target the market 50th percentile).

Payable Date

The date that a corporate distribution, such as a dividend, is payable to the record holders of a corporation's securities.

Payable Time

Time that is ready to be collected by the payroll system (see Time Reporting).

PayCycle

A set of rules that define to PeopleSoft Payables the criteria by which it should select scheduled payments for payment creation.

Payee

Any payroll recipient. A payee can be an employee or a non-employee of an organization.

Payee Process Stat Record

A record created for each payee during the payroll process. The system creates one Process Stat record per payee for each calendar.

Payee Section

Type of section that can be added to a process list. A payee section defines a set of elements that is to be resolved for a particular payee.

Payline

Record containing standard payroll information for an employee, such as the amount of regular pay, number of regular hours, additional pay (if any), and tax information and job data.

Payment Interface

An Application Engine process that loads payment information from the Banks Statement tables and the Staging tables to the Application tables and performs various checking and default operations.

Payment Predictor

PeopleSoft Receivables' automatic cash application feature that pairs open items with unapplied payments based on predefined algorithms.

Payment Schedule

A schedule of payment dates for leased assets.

Payment Selection

A process by which PeopleSoft Payables selects scheduled payments that are eligible to be paid in a pay cycle.

Payment Worksheet

The work space in which open items are paired with unapplied payments.

Payroll Certifying Officer

The individual with the delegated authority for approving all items relating to payroll for those employees under his/her authority.

Payroll Process Tables.

Records holding data necessary to process a payroll, such as employee, company, and tax information.

Paysheets

Repository for the raw data necessary to calculate pay for employees, including earnings, hours, deductions, taxes, and accounting data.

PBGC Rates and PBGC Grading

The interest rates published monthly by the Pension Benefit Guaranty Corporation. There is an "immediate" rate that applies once benefits commence as well as a series of "graded" rates—calculated based on the immediate rate—that are used during the time between benefit determination and a deferred benefit commencement.

Pegged Chain

A method the PeopleSoft Enterprise Planning and Production Planning solvers use to determine feasible plans. The method ties tasks together in order to explicitly record which supplies are being used to satisfy which demands. Through this process, the Planning engine first determines which independent demand has the highest priority. Then, it determines the lateness preference ranking for dependent demand tasks.

Penalty

A user-assigned value for constraints that can be violated, determining how the schedule's score will be calculated. Setting the penalty configures the constraint to your priority. Use the Control Page to assign a higher penalty to violations that are more critical to your schedule or a lower penalty to constraints that you can deal with externally.

Pending Exception

Any known exception to an employee's scheduled workday. Pending Exceptions are future dated (future is defined to be for a date under report beyond the last date of the employee's current pay period).

Pending Item

Information in PeopleSoft Receivables that has been entered in or created by the system, but hasn't yet been posted. During the Receivable Update process, the system uses the pending items to update customer balances—either by creating new items or by adding item activity lines to existing items.

Pending Time

Time that has been reported or is assumed to have been reported (based on employee work schedule and calendar date) that has not been used by the business entity. Pending Time may be for past, current, and future pay periods. It is the label for those time transactions that are waiting to be used by the business (for example, approved and unapproved time not yet updated to Paysheets).

Pension Status

An employee's standing with regard to a particular pension plan. For example, employees can be active participants, terminated deferred vested, or in pay status.

PeopleCode

PeopleSoft's proprietary language; it is executed by the PeopleSoft Application Processor. PeopleCode generates results based upon specific actions, based upon existing data or the

actions of a user. Business Interlink Objects are executed by calling the execute() method from PeopleCode. This makes external services available to all PeopleSoft applications wherever PeopleCode can be executed.

PeopleCode Event

An action that an end-user takes upon an object, usually a Record Field, that is referenced within a PeopleSoft page.

PeopleSoft Activity-Based Management (ABM)

A PeopleSoft Analytic Application that aligns organizational costs with operational activities, enabling a coordinated approach to expense and PeopleSoft Activity-Based Management. PeopleSoft Activity-Based Management identifies and assigns operational activities to products, customers, or services.

PeopleSoft Analytic Applications

These are applications within Enterprise Performance Management (EPM) that help you enrich the data in the PeopleSoft Enterprise Warehouse and perform forward looking simulations and scenarios. These applications include: PeopleSoft Activity-Based Management (ABM), PeopleSoft Asset Liability Management (ALM), PeopleSoft Balanced Scorecard (BSC), PeopleSoft Funds Transfer Pricing (FTP), PeopleSoft Risk Weighted Capital (RWC), and PeopleSoft Workforce Rewards. PeopleSoft Funds Transfer Pricing and PeopleSoft Risk Weighted Capital are applications that target the financial services industry (FSI).

PeopleSoft Asset Liability Management (ALM)

PeopleSoft Asset Liability Management provides financial service institutions with the analytical tools to define, measure, monitor and manage interest rate risk, liquidity risk, options risk, and to some extent exchange rate risk. The primary audience for PeopleSoft Asset Liability Management is the financial institution's Asset/Liability Committee (ALCO).

PeopleSoft Balanced Scorecard (BSC)

PeopleSoft Balanced Scorecard converts an organization's vision and strategy into a comprehensive set of performance and action measures that provide the basis for a strategic management system.

PeopleSoft Budgeting

A budgeting application that is a combination of Education and Government (E&G) Budget Planning and Budgets (commercial). This application resides on the EPM database, primarily using the ODS layer of the PeopleSoft Enterprise Warehouse for its data.

PeopleSoft Business Analysis Modeler (BAM)

A multi-dimensional modeling tool used to support several analytic applications.

PeopleSoft Business Planning

A planning application that enables financial executives to model various alternatives and set corporate financial targets to achieve their strategic goals. PeopleSoft Business Planning integrates with PeopleSoft Analytic Forecasting, PeopleSoft Activity Based Management, PeopleSoft Workforce Analytics, and PeopleSoft Budgeting applications.

PeopleSoft Customer Behavior Modeling

A PeopleSoft application that enables you to: a) create a customer profile by extracting customer data from the Enterprise Warehouse; b) create segments and samples from the profile to efficiently target marketing campaigns and further analyze customer behavior; c) append to it additional data from external sources such as demographic, credit or psychographic information; d) use a data mining tool to create a predictive model; e) score the customers in your profile using the predictive model. You can then publish the results to another transactional application.

PeopleSoft Customer Scorecard

A product that provides a pre-defined set of customer-oriented key performance indicators (KPIs), to help you build a scorecard specific to your organization. This scorecard facilitates the measurement and communication of customer satisfaction, customer activity, and objectives across your organization.

See also PeopleSoft Balanced Scorecard

PeopleSoft Enterprise Performance Management (EPM)

Enterprise Performance Management is a comprehensive, integrated analytic business solution designed to increase the value of organizations by enabling people to make better decisions. The PeopleSoft Enterprise Performance Management product line consists of the PeopleSoft Enterprise Warehouse and optional analytic applications and Data Mart products.

PeopleSoft Enterprise Warehouse (EW)

PeopleSoft's data warehousing solution. The PeopleSoft Enterprise Warehouse provides the tools necessary to query, analyze, and present information to provide the optimal environment for business intelligence. It is the central repository for data that will be used with the analytic applications in the Enterprise Performance Management product line, and can also serve as a standalone data warehouse. The PeopleSoft Enterprise Warehouse consists of dimension, fact, reference, and error tables, reporting and ETL tools (Informatica PowerMart), as well as the Operational Data Store (ODS). The tables in the PeopleSoft Enterprise Warehouse are maintained separately from your transaction-based systems to allow for comprehensive analysis of data originating from any Online Transaction Processing (OLTP) or legacy system.

PeopleSoft Funds Transfer Pricing (FTP)

A PeopleSoft Analytic Application that enables an institution to accurately measure and tune profitability. PeopleSoft Funds Transfer Pricing is an interest rate that represents the value of an asset or liability to the institution. PeopleSoft Funds Transfer Pricing is based on market rates, adjusted for risk and cost variables, specific to the institution. By assigning PeopleSoft Funds Transfer Pricing to each item on the balance sheet, the institution can remove the

effects of interest rate volatility from business units, so that profitability measurements are based on factors within their control, that is, credit quality, pricing and product strategy.

PeopleSoft Operations Data Stores (PODS)

See Data Warehouse Tables

PeopleSoft Marketing Insight

A tool that helps you analyze your marketing campaigns and activities. It helps you determine the effectiveness of marketing events based on factors such as number of leads generated, profiles of respondents, campaign return on investment, and campaign forecasted costs to complete.

PeopleSoft Risk Weighted Capital (RWC)

A PeopleSoft Analytic Application that enables the financial services industry to accurately measure capital that has accounted for risk. RWC allocates capital to various levels within a financial institution according to risk, providing the opportunity to measure performance based on how well each business unit, product, customer, or transaction generates income given its perceived level of risk as quantified by the allocation of capital.

PeopleSoft Sales Activity Insight

A tool that helps you analyze key components of the sales process, such as pipeline status, discount analysis, and sales process.

PeopleSoft Support Insight

A tool that helps you determine the effectiveness of your customer service organization. It helps you answer questions such as: Are we effectively handling customer issues? Has contact center performance changed from last year to this year? Which product quality issues are most prevalent?

PeopleSoft Workforce Analytics (WFA)

PeopleSoft's complete workforce analysis solution, which includes the PeopleSoft Workforce Rewards analytical application and the PeopleSoft Workforce Analysis Insight. The complete solution set includes the PeopleSoft Enterprise Warehouse and PeopleSoft Balanced Scorecard products. It helps to manage strategic employee compensation, goals, and competencies, as well as retention.

PeopleSoft Workforce Rewards

PeopleSoft Workforce Rewards is an analytical application you use to align your workforce compensation and retention initiatives with the strategic objectives of your organization. Modules include Market Compensation, Base Pay Structure, Compensation Planning, Workforce Simulation, and Retention Management. You integrate data from multiple internal and external sources, enrich it using rules you define based on any data in your PeopleSoft Enterprise Warehouse, and simulate multiple scenarios of future workforce compensation and

retention activity. You then analyze and evaluate your scenarios, and make actionable decisions you can communicate back to your PeopleSoft ERP systems for execution.

PeopleSoft Portal

The portal bundled with every PeopleSoft 8 application. It provides a simple navigation system, based on existing menu definitions that have been imported into the portal registry. Navigation to content outside of PeopleSoft applications is not provided.

Percent Cycles Without a Shortage

In PeopleSoft Inventory Planning, a method used with safety stock policies. The value is derived from the percentage of replenishment cycles that will complete without a stockout.

Percent Demand Fill

In PeopleSoft Inventory Planning, a method that can be used with safety stock policies. This method defines the percentage of the total quantity ordered that must be filled without a backorder.

Percentage Tolerance

The acceptable percentage difference between expected cycle count quantities and actual quantities counted in PeopleSoft Inventory. This value provides a margin of error for an item during cycle count reconciliation count quantities.

Performance Appraisal Code

Indicates the level of performance of an employee.

Performance Appraisal Due Date

Date established based on the WGI or LEI for the yearly appraisal of an employee.

Period Closing Offset

In Time and Labor, the closing date beyond which this pay period is not considered current any longer, if the period's closing date is different from its end date. You can enter a positive or negative number of days.

Period Method

In PeopleSoft Inventory Planning, a method used to determine how a single static policy value is to be calculated from time-phased results with static policies.

Period of Interest

The maximum period of time containing the data needed to run all the rules in a rule program (see Batch Processing)

Period Segmentation

When an element (like compensation rate) changes mid-period, requiring all other elements in the process list to be calculated multiple times on either side of the date on which the change took place, period segmentation is used. The system calculates each element more than once, using the components that were effective during the different time slices. The system keeps the results of these calculations separate with the object of creating two gross-to-net result sets.

Periodic Processing

In PeopleSoft Pension Administration, any of several batch processes that a plan administrator must run on a regular basis—for example, consolidation of payroll data.

Personal List

A user-created list of products defined in PeopleSoft eStore, used to quickly populate the shopping cart when creating a new order in either PeopleSoft eStore or Mobile Order Management.

Personal Register (Registre Unique du Personnel)

In France, companies are required to be able to produce, at any given time, a Personal Register. For a given establishment, this report lists current employees and employees who left up to 5 years ago.

Personnel Action

Personnel actions are changes to employee data or status resulting from such activities as promotions, transfers, terminations, salary increases, and leaves of absence.

Personnel Representatives (Délégués du personnel)

In France it is mandatory for companies with more than 11 employees to elect personnel representatives who will represent all of the employees before management.

Perspective

In PeopleSoft Balanced Scorecard, a category for organizing critical success factors and key performance indicators. Usually there are four: financial, customer, internal process, learning and growth.

PF Ledger

The PF Ledger (PF_LEDGER_F00) is an important fact table within the PeopleSoft Enterprise Warehouse. The primary function of the PF Ledger table is to support PeopleSoft Enterprise Performance Management reporting. The data that gets posted to the PF Ledger must be accurate and clean.

PF Business Unit

PF Business Units differ from other PeopleSoft Business Units in that they represent functional or strategic areas of an organization, rather than separate legal entities.

Phase Type

A label for the different phases you want to define for a project. Costs can then be calculated by project phases. Examples include planning, startup, construction, and cleanup.

Physical Accounting

The PeopleSoft Inventory feature that updates tables based on count result input, regardless of how the count was created or the data collected.

Physical Inventory Process

In PeopleSoft Asset Management, the process by which you extract asset data from the Asset Management database to load into your bar code scanning device. You then scan the assets and load the data gathered during the actual physical inventory into Asset Management, enabling you to generate physical inventory results for review. You perform matching and generate transactions to reconcile the data in Asset Management with the results of your physical inventory.

PIA

PeopleSoft Internet Architecture. This is the fundamental architecture on which PeopleSoft 8 applications are constructed, consisting of an RDBMS, an application server, a web server, and a browser.

Piece Work

Method of compensating time reporters based on units completed rather than hours worked

PIN

Technical term for an element. In PeopleSoft Global Payroll, PIN is often referred to in the online object names and within the batch code. PIN stands for Pay Item Name.

Placeholder

A temporary location designator in an engineering bill of material for a component item that has yet to exist. These temporary placeholders have to change into approved items before transferring engineering bills of material (EBOM) to manufacturing bills of material (MBOM).

Plan Administrator

The person selected by the employer to perform the administration of a plan under PeopleSoft Pension Administration.

Plan Eligibility

The PeopleSoft Pension Administration function that uses job data to determine whether an employee may participate in a pension plan. An employee can be eligible based on job data but not be participating because of an unmet service or age requirement.

Plan Type

A unique ranges of codes used during payroll calculation to determine deduction processing rules. See also Benefit Plan Type.

Plan Year

The annual period that a pension plan uses to measure service, earnings, and benefits. Generally, the pension plan year will match the fiscal year of the plan sponsor.

Planning Item

A non-inventory item designated as being used for planning purposes only. It can represent a family or group of inventory items. It can have a planning bill of material or planning routing, and can exist as a component on a planning bill of material. A planning item cannot be specified on a production or engineering BOM or routing, nor used as a component in production. Quantity on hand will never be maintained.

Planning Level

The level ona dimension's tree used for planning. Typically, a customer will choose not to plan at the lowest level of available detail, such as the individual product level. Instead, the individual products are mapped to their corresponding product group and the planning is done at the product group level.

Planning Targets

The amount the budget must equal, such as a budget spending limit or cap for expenses where users can not exceed the amount. Planning Targets are presented at a summary level. This term is interchangeable with Spending Limits.

Planning Target Tolerance

The percentage and/or amount a user can be over or under the planned budget target.

Planning Target Control

Values set at the user role level, including no control, active control, and passive control. For more information see no control, active control, and passive control.

Planning Task

Any activity in PeopleSoft Planning that creates a schedule.

PODS (PeopleSoft Operations Data Stores)

See Data Warehouse Tables

POI. Personnel Office Identifier.

Also known as Submitting Office Number (SON). These are codes assigned by the OPM to the office(s) delegated authority within an agency to process personnel actions on Federal employees.

Pointers

A pointer is an "address" of a driver quantity, or value, within the Enterprise Performance Management product line. Pointers are used as a means of defining where driver quantities exist in tables that reside in the PeopleSoft Enterprise Warehouse. Pointers enable you to extract values from any location in the warehouse and then use these values as driver quantities. There are three different kinds of pointers: explicit, implicit, and multidimensional.

Policy Control Group

In PeopleSoft Inventory Planning, a feature for setting up order quantity, safety stock, reorder quantity, and maximum and minimum policies. The **Control Group** is assigned to a set of **Planning Item**. The policies of the associated planning items can be set explicitly or defaulted from the policies on the policy control group.

Policy Generation

In PeopleSoft Inventory Planning, a set of run options used to control the functions and behavior of the Policy Generation program.

Policy Item

An item record which is related to a location and for which **Inventory Policy** is held. A policy set, **Planning Item** ID, and Location ID uniquely identify a policy item. The combination of an item and a location is called a stockkeeping unit.

Policy Set

Defines a set of the items for which **Inventory Policy** is to be calculated. Each policy set is assigned a unique ID and includes information that defines, for example, the associated **Forecast View**, time periods, and planning horizon.

Policy Simulation

In PeopleSoft Inventory Planning, a feature that simulates the effects of various stocking scenarios, compares current policy with simulated policy, and determines the best inventory investment strategy.

Population

A Population is defined on top of DataMaps created using Enterprise Warehouse metadata. The Population builder allows you to easily format a SQL statement, using filters, to select

rows from one or more tables joined together in a DataMap. You can view the results of a Population directly from the browser.

Pop Up Payment Option

A variation on a joint and survivor payment option under which the benefit payable to the participant is increased if the beneficiary should die prior to the participant.

Portal

A portal is a web site that helps you navigate to other web-based applications and content. Users often consider a portal their "entry point"—the place they typically visit first after launching their web browser.

Portal Registry

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.

Portal Registry API

The Registry API is provided for accessing each portal registry from PeopleCode, COM, Java, or C programs. Providing the same kind of registry management capability as the online administration pages, it can be used by external systems to update the registry to reflect changes in the content reference URL, taxonomy, and effective dates. The Registry API is fully described in the PeopleCode documentation.

Portal Servlet

A Java servlet that runs on a web server. The portal servlet intercepts user requests for content, retrieves content, and builds a single HTML document to be displayed in the user's browser.

Portal Solutions

Portal Solutions are separate product offerings from PeopleSoft that consist of pre-built, packaged solutions focused at different audiences (customers, suppliers, and employees). Because they are both pre-built, supported application products, Portal Solutions can be deployed swiftly and easily, saving significant resources when compared to other custom-built solutions.

Position

The officially assigned duties and responsibilities that make up the work performed by an employee. Positions are linked to Job Codes, which can be considered the electronic version of the Position Description. There can be a many-to-one relationship between the Position and Job Code.

Position Budgeting

The budget amounts (salary, benefits, and earnings) associated with positions within an organization. Position budgeting can be calculated based on position information loaded from a human resource system. Position budgets are used to generate line-item budgets for personnel costs. PeopleSoft Budgeting-specific.

Position Change

A move by an employee to another position during the employee's continuous service under the same appointment within the same agency.

Position Date Created

Date the position was created for use in the agency.

Position Description (PD)

In accordance with OPM guidelines, an official description, authorized and approved by an agency official, describing duties and responsibilities to be performed. Position classification standards are used to describe the work, classify the work components by occupational series, and factors (e.g. supervisory control, scope, complexity, competencies required) are used to determine the grade level (i.e., salary range) for the position.

Position Description Number

A number assigned to identify various types of Position Descriptions.

Position Description Required

Identifies those positions for which a position description must be maintained.

Position Number

A number that identifies an authorized Position.

Positive Input

Data such as hours worked or a bonus amount entered for elements that change each pay period. Positive input can be entered manually, generated by the system, or received from other applications.

Positive Task Reporting

A method of time reporting in which all required task elements must be provided (see Time Reporting)

Positive Time Reporting

A method of time reporting in which all elements of time must be provided (see Time Reporting)

Post Differential Percent

Additional compensation that may be paid to certain employees who work in Guam or the Northern Mariana Islands.

Post Differential, Non-Foreign

A differential payable to an employee at a location in a non-foreign area if conditions of environment differ substantially from conditions of environment in the contiguous United States and warrant its payment as a recruitment incentive.

Post-56 Military Deposit

The OPM provides guidelines to Federal agencies on how to calculate and process these voluntary employee deductions from pay toward the employee's current retirement fund for those periods of eligible military service.

Posting

In PeopleSoft the process by which accounting entries are created or updated based on user transaction input and accounting entry templates. In PeopleSoft Receivables, posting is also known as Receivable Update.

These posted accounting entries in the feeder systems,such as accounts receivable or accounts payable, must be further processed by the Journal Generator to create journal entries before posting to the General Ledger occurs.

Post-Tax Deductions

See After-Tax Deductions

Pre-encumbrance

An encumbrance that occurs before an employee/employer relationship exists. You encumber funds for an employee you have on staff; you pre-encumber funds for an employee that you anticipate hiring. For example, you would pre-encumber funds for a new position that has just been approved but not filled.

Pre-encumbrance Ledger

Stores pre-encumbrance amounts. Updated by posting pre-encumbrances, such as purchase- or hiring-requisition source transactions (including journal entries).

Premium

Any additional compensation for extra hours worked, often expressed in terms of factor-above-normal-per-hour pay, such as time and a half (where one-half is premium pay), double time or triple time. Also, any additional pay provided to a time reporter based on compensation rules (see Time Administration in your *PeopleSoft Time and Labor* PeopleBook).

Prenote

A prenotification or waiting period requested by banks before processing payroll direct deposits.

Pre-Retirement Survivor Annuity (PRSA)

A benefit paid to a beneficiary if a pension plan participant dies before commencing benefits. Qualified plans must offer a pre-retirement survivor annuity, although the employee can be required to pay for the coverage with a reduction in the benefit.

Pre-Tax Deductions

See Before-Tax Deductions

Previous Day's Close

The previous trading day's last reported trade.

Price Break

Defined in PeopleSoft Order Management, and linked with price sets, a price break defines the actual adjustments that are made to the list price. They are valid only within a time frame you establish.

Price Rule

Used in conjunction with price sets, rules are essentially a decision tree that establishes the search order the system uses in reviewing sets for a match on the variables they reference for price adjustments.

Price Set

Linked with price breaks, a price set specifies the parameters for your price adjustment. It establishes selection criteria, determines whether the break is based on quantity or price, and defines how the adjustment is applied.

Price Source

A service provider or publication that reports the trading activity for a stock traded on stock exchanges. Examples include Wall Street Journal and Bloomberg.

Primary event code

Primary event codes, also called purpose codes, specify the status of the transaction: whether it's a new transaction, a cancellation, a duplicate, a status request, and so on. Every transaction has a primary event code assigned to it.

Primary Insurance Amount (PIA)

The benefit amount calculated under the Social Security benefit formula.

Primary scroll record

Primary scroll records are the principal SQL table or view associated with a page scroll level. A primary scroll record uniquely identifies a scroll level in the context of its page: each scroll level can have only one primary scroll record; and the same primary scroll record cannot occur on more than one scroll at the same level of the page.

Prior Period

In Time and Labor, any payroll period before the current one.

Prior Period Adjustment

A change or correction to previously reported time or task information, or an insertion of time or task information. Often requires the original report to be offset (reversed) and the correct information to be recorded. (see Time Reporting)

Priority Rank

The numeric value assigned to inventory **Demand Priority Rules**. The lower the number, the higher priority of the rule and the orders matching that rule.

Private

A tracking method used by a privately held company to track their daily prices. The Board of Directors typically establishes a price for a period of time. Stock of a privately held company is not traded on an exchange.

Private Views

User-defined views available only to the user who created them. For more information, *see* Budget Views.

Process

See Batch Processes.

Process Definition

Process Definitions are created in the Process Scheduler Manager pages to define each specific run request. A Process Definition is comprised of a variety of variables including panels associated with a request, Process Groups, output parameters, run locations, and many more.

Process Group

Used to associate specific Process Definitions with a Class Profile in Security Administrator. This allows administrators to restrict an operator's ability to initiate requests.

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

Multiple Process Definitions can be logically linked into a job request to process each request serially or in parallel, and optionally initiate subsequent processes based on the return code from each prior request.

Process List

The set of instructions the system uses during a payroll process to determine which elements to resolve. A process list is comprised of sections that identify the sets of elements to be resolved. You build process lists and attach them to calendars.

Process List Manager

The program used during batch processing that reads the Process List and calls the PIN Manager to resolve elements on the list.

Process Request

A single "run request," such as an SQR, a COBOL program, or a Crystal report that you run through Process Scheduler.

Process Run Control

A PeopleTools variable used to retain Process Scheduler-defined values needed at runtime for all requests referencing a run control ID. This is not to be confused 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.

Process Scheduler

A PeopleTool that performs tasks behind the scenes of your application. It can run several kinds of processes, such as COBOL, SQR, and Application Engine programs. You can schedule processes to run on a regular schedule or at your request. Processes can run on your workstation or on a server.

Process Scheduler Server Agent

The server-based program (PTPUPRCS) that manages the selection, validation, and initiation of all queued requests for each defined server within your batch environment (Process Scheduler).

Process Type

A global process definition under which related process definitions are grouped. This allows for easy maintenance of Process Definitions that share common parameters.

Processing group

In order to partition application processing between the client and the application server, it is necessary to define units that, as a whole, run in one location or the other. We call these units processing groups.

Processing groups can encompass one or more PeopleCode events. Some processing groups can run either on the client or on the application server, such as Component Build and Component Save.

Product

A commodity primarily defined in PeopleSoft Order Management. It may be: 1) The Order Management view of an inventory item that has attributes the same as or different from those of its inventory counterpart. 2) A commodity that is not a stocked inventory item such as a product kit or a service. 3) A tangible commodity that is drop shipped from another vendor and is never stocked in inventory.

Product Alternate

Alternative products that can replace the product ordered when it's out of stock or a problem with a particular product temporarily prevents shipment.

Product Catalog

A configurable list of available products that may be of interest to a specific customer. You can create two types of catalogs—inclusive catalogs that contain all the products you want made available to a customer and exclusive catalogs that contain the products you want to withhold from your customer. By attaching product catalogs to a Sold To customer, you define the products they can buy.

Product Definitions

This support module describes how other support modules process the instruments that belong to a particular product. For example, different products have different cash flow characteristics and may be stratified differently, or may react differently to changes in interest rates. This module enables you to specify each of these actions for each instrument.

Product Kit

A commodity that consists of a fixed set of components that are sold together. It appears as a single line on an order, but is represented by multiple lines on shipping documents. Product kits may comprise inventory items, non-stockable products, or a combination of both.

Product Kit Component

A commodity that is part of a product kit. It may be an inventory item or a non-stockable product such as a service.

Product Pricing Model

In the financial services industry, this defines models that describe indices upon which future rates are paid or charged for an individual product.

Product Tree

A user-defined graphical representation of a company's product structure. A product tree defines how products are promoted and determines what users have authority to promote those products.

Production Maintenance Spreadsheets

A set of spreadsheets generated by an nVision process, containing production ID and production schedule data extracted from PeopleSoft Production Management. You can add and maintain production quantity data using these spreadsheets and then import the data back to Production Management.

Production Option

Effective-dated combinations of BOM codes and routing codes. You can create multiple effective-dated BOM code/routing code combinations (or production options) for an item. These combinations can be extracted to PeopleSoft Production Planning. They enable the specification of multiple production variations for an item and provide control of seasonal variations by effective date.

Production Option Cost

A cost based on a specific BOM/routing combination (also known as a production option). The Production Option Cost utility rolls up production options costs based on specific BOM/routing combinations, enabling you to cost individual production options and later to have the DataLink pass this cost to the Planning engine. If this utility isn't run, the Planning engine will use the standard item cost based solely on the primary BOM and routing instead.

Productive Time

Employee scheduled time spent performing any task for which a position was created; work performed on behalf of a business entity that is required for that entity to fulfill its business purpose. Employees doing the work they or someone else was hired to do.

Productive Unit

In Italy employers organize employees into productive units based on agreements between the unions and the employer.

Profile

A data set that you aggregate from the Enterprise Warehouse, according to the filters you specify, the Key Performance Indicators you select, and the 3rd party demographic data you include.

Profile Factor

In PeopleSoft Demand Planning, the weight index assigned to each **Forecast Period** to take into account seasonal fluctuations in demand. The factor or index typically measures the percentage of difference between the base demand and the expected actual demand in the period.

Profiles

Group of employees defined according to a list of job codes and departments. You can use these profiles to ascertain training demands within your organization based on set criteria.

Profit Manager

The Profit Manager is a set of integrated tools that enable true profitability reporting. Profit Manager features are tightly integrated with the PeopleSoft Analytic Applications and provide you with ways to ensure data integrity, edit data, and post data to the Performance Ledger table.

Project

The highest level of hierarchical organization within PeopleSoft Projects. Projects provide the structure to which activities and resources are added. Each node on a Projects tree represents a project. Projects can contain other projects as well as activities and resources. This provides a hierarchical relationship between projects and facilitates cost roll-ups.

In Enterprise Performance Management you use a Project to create or modify a Profile. A Project contains pointers to data elements that you include in a Profile.

Project

In PeopleSoft Time and Labor, a specific endeavor undertaken to achieve a specific goal. Typically, projects are approved and undertaken with level of cost, schedule, and performance already agreed upon. A project is composed of a set of tasks, each of which requires staffing, provisioning, and/or scheduling. Project progress is often measured in terms of task completion.

Project ID

The name or number by which a project is to be identified in all tables and pages.

Project Type

A user-definable grouping of projects. Project types are optional.

Projected Run Date

In PeopleSoft Demand Planning, a calculation made that projects a life volume for a period based on a calculated run rate or performance ratio.

Projection.

An estimated pension benefit calculated as of a future date or any estimated data used as the basis for such a calculation.

Promotion

- For positions under the same type job classification system and pay schedule, a promotion changes the employee to a higher grade level or makes permanent a Promotion NTE;
- When the old and new positions are under different job classification systems and pay schedules, a promotion changes the employee to a position with a higher rate of basic pay or makes permanent a Promotion NTE.

Promotion Pattern

In PeopleSoft Demand Planning, an **Event** function that enables you to apply weights to promotions across a range of **Forecast Period**.

Prompting Profile

A task profile usually used by account managers as a way of creating task profiles for employees who report task time differently by customer. For example, you might have an account manager who has fifty customers; when the account manager comes in each day to report time, the system will display all the customers, and indicate which customers it will use as a default if she doesn't manually report time.

Prorated

In Enterprise Planning and Simulation, prorated is when the computed forecast and the summarized forecast are two different versions of the statistical forecast. In addition, the forecast at the product family level can be allocated down to the individual products. Usually this allocation is done in proportion to the calculated product forecasts at that level. This version of the (statistical) forecast is called the allocated or prorated statistical forecast.

Pro-Rate Purchase

A purchase in which the number of shares to be purchased is prorated according to a specified factor. This may occur when the total number of shares to be purchased is greater than the number of shares allocated to the stock plan from the treasury.

Prorated Forecast

In PeopleSoft Demand Planning, a forecast developed by factoring the group forecast down one level at a time to make the sum of the item forecast equal to the aggregate forecast. The prorated forecast tends to be more accurate than the **Statistical Forecast**.

Proration Rule

Element that defines how you want to prorate an item. You use proration rules in numerous places—for instance you could prorate an earning, deduction, or many of the elements that make up an earning or deduction.

Proration Threshold Ratios

In PeopleSoft Demand Planning, the upper and lower ratios used as thresholds for **Reasonableness** checks when a forecast is developed using proration.

Provider

An entity, such as an insurance company, that provides one or more of the benefits your company offers. For example, Metropolitan Life Insurance Company is a provider to companies that use a Metropolitan life plan.

Proxy Person

A highly compensated executive. Corporations must include information regarding the most highly compensated executive officers in their proxy reporting.

Proxy Statement

The document that must accompany a solicitation of proxy appointment under SEC regulations. The purpose of a proxy statement is to provide shareholders with the appropriate information to make an intelligent decision.

PSADMIN

A PeopleSoft utility providing a menu interface to create, configure and administer application server domains and the Process Scheduler Server Agent (Batch Server).

PSADMIN

A PeopleSoft utility providing a menu interface to create, configure and administer application server domains and the Process Scheduler Server Agent (Batch Server).

PSAPPSRV

PSAPPSRV is the main server process running within a domain. PSSAPPSRV performs the functional requests, such as building and loading components. It also manages the memory and disk-caching for PeopleTools objects on the application server. Each PSAPPSRV process maintains its own memory and disk cache.

It provides authentication services for incoming users. For instance, it checks the PeopleSoft OPRID against the directory server or PSOPRDEFN table.

PSQCKSRV

Essentially, PSQCKSRV is a copy of the PSAPPSRV. It performs quick, read-only SQL requests. It is an optional Server Process designed to improve performance by handling items in the PSAPPSRV transaction request queue.

PSQRYSRV

Like the PSQCKSRV server process, PSQRYSRV is designed to alleviate the workload of PSAPPSRV. PSQRYSRV is designed to specifically handle all user-generated queries submitted by PeopleSoft Query (PSQED.EXE). This server process is designed to improve overall application server performance whether or not you have PSQCKSRV configured. It is specifically, and exclusively designed to process PeopleSoft Query transactions, which can be very SQL intensive.

PSSAMSRV

It processes conversational SQL transactions primarily for Application Designer.

Public Company

A company that has held an initial public offering and whose shares are traded on a stock exchange or in the over-the-counter market. Public companies are subject to periodic filing and other obligations under the federal securities laws.

Public Views

Coordinator-defined views, available to anyone using the application. For more information, *see* Budget Views.

Publish/Subscribe

Publish/Subscribe type messaging is performed with PeopleTools Application Messaging technology. You can send data from one PeopleSoft system to another in an asynchronous mode—meaning the two systems don't have to be sending and receiving at the same time. This is possible because the message transfer is accomplished through a Web server with an "http: gateway."

Pull List

Similar to a pick list, a pull list contains multiple replenishment requests, including the location, quantity, and item quantity required in a specific sorting sequence. You use pull lists in PeopleSoft Flow Production with Inventory replenishment.

Pull Ticket

A document containing the details of a single request replenishment request, including Kanban ID, item, quantity, source, and To locations. You use pull tickets in PeopleSoft Flow Production with Inventory replenishment.

Punch

Precise instances of date and time recorded for a user and measured in seconds, minutes, hours, day, month and year and time zone (see Time Reporting)

Punch Duration

Length of time between two punches in increments of hours or partial hours (see Time Reporting)

Punch Matching

Area of the application which converts paired punches to punch duration by processing rounding rules and assigning the tasks to the appropriate logical day based on rules established by the user

Punch Restriction

The facility to constrain a time reporter's ability to create a punch that deviates from the schedule (see Time Reporting)

Punch Type

A user defined classification of punches, i.e. In, Out, Start, Stop (see Time Reporting)

Purchase

The issuance or purchase of shares through a stock purchase program. The purchase is made using current contributions from a participant and any carry-forward remaining for the participant from previous purchases.

Purchase Price

The discounted price paid for the shares at the end of a purchase period.

Purchase Price Variance

A PeopleSoft Payables matching feature that compares purchase order and inventory prices for any variance in the prices.

Purge Rules

The rules that define criteria to clear data you no longer need from previous open enrollment processing cycles in PeopleSoft Benefits Administration.

Pyramiding

A computer calculation enabling an individual owner of one share of stock to use the stock-swap technique to exercise a stock option of any size without using cash. Not many corporations permit pyramiding.

Q

QDRO

See Qualified Domestic Relations Order.

QDRO Alternate Payee

A former spouse who is entitled to a portion of a participant's pension benefits as a result of a court order.

QJSA (Qualified Joint and Survivor Annuity)

A post-retirement death benefit for a spouse. Plans subject to this requirement must provide an annuity for the life of the participant with a survivor annuity for the life of the participant's spouse.

QMCSO (Qualified Medical Child Support Order)

A QMCSO is a court order that requires a group health care plan to provide benefits to the child of a participant as part of a child support arrangement on the behalf of that participant. Base Benefits enables the tracking of QMCSOs for dependents.

Qualified Domestic Relations Order (QDRO)

A court order ordering a division of a participant's pension benefits. This is normally the result of a divorce and gives a portion of the pension benefits to the former spouse.

Qualified Plan

A pension plan for which the employer can take tax deductions for contributions to the plan. Investment income of the plan trust fund is not taxable to the employer. Tax law places restrictions on the plan rules.

Qualifying Dispositions

A transaction whereby a participant sells shares acquired through a stock purchase plan two years after the grant date and one year after the purchase date.

Quality Function

A level of configuration that enables you to define the fields and attendant information that provides a base level for inspection plan and integration. Quality functions enable you to map process-specific field information into PeopleSoft Quality for identification, tracking, and analysis.

Quality Ranking Factors

Knowledge, skills, and abilities that could be expected to enhance significantly performance in a position, but are not essential for satisfactory performance. Applicants who possess such

KSAs may be ranked above those who do not, but no one may be rated ineligible solely for failure to possess such KSAs.

Quality Server

A PeopleTools-based analysis and graphing program.

Quality Step Increase (QSI)

A step increase awarded to an employee for sustained high quality performance.

Quantity Allocation Method

In PeopleSoft Inventory, the method used to determine how available quantity will be allocated to prioritized demand lines when using the online reservations page.

Quantity Precision Rules

A set of rules specifying whether item quantities for a given unit of measure are expressed as whole numbers or as decimals. Quantity precision is set at the inventory **SetID** and item-UOM levels.

Query

A set of data members that are selected from a Class catalog (provided by the Business Interlink Plug-in) as well as a generic form of Criteria. The criteria are composed of <left-hand-side> <Relational Operator> <right-hand-side> statements that can be concatenated using a set of logical operators. All operators and class catalogs are dynamically provided through the Business Interlink Plug-in.

R

Race And National Origin Code

A code that identifies the employee's basic racial and national origin category.

Range of Dates Reporting

A Time and Labor process that enables you to report a start and stop date, a time reporting code and task information for a single employee. The system transforms the information into instances of daily time based on the employee's schedule or default work schedule, replacing the scheduled time with the entered Time Reporting code and the number of scheduled hours on a day-to-day basis.

Range Penetration

In PeopleSoft Workforce Analytics, Range Penetration is the degree to which an employee's actual pay has progressed through their salary grade, and is expressed as a percentage. The calculation is:

Range penetration = (Employee Base Pay – Range Minimum)/(Range Maximum – Range Minimum).

Range Width

In PeopleSoft Workforce Analytics, the difference between the maximum and the minimum values of the pay range calculated using the following formula (and expressed as a percentage):

(Maximum – Minimum)/Minimum.

Rapid Time Entry

The process that enables you to enter daily time for single employees without the system editing your field entries. The system populates temporary tables, which are used by a batch process that reads, edits and moves the data into the appropriate time and labor tables. You cannot prompt for valid values in any of the fields, and the online system does not edit any of the data you enter against other tables.

Rate Code

Alphanumeric reference to the cost per hour or unit of time reported to a specific TRC.

Rate Code [Global Payroll]

IDs for pay components. Rate codes define rates of pay and are set up in the Comp Rate Code table. Rate codes are then used to represent pay components in pages and when you configure compensation packages in the compensation record.

Rate Code Group

A rate code group is a number of pay components (represented by rate codes) bundled into a subset of a compensation package. The rate code group is used to calculate percentage-based components that do not apply the percentage to all pay components in the compensation package. Rate code groups are constructed on the Rate Code Groups page.

Rate Code Type

Defines how the monetary value of the rate code is calculated. The compensation rate code type is defined on the Comp Rate Code table. Valid values are Flat Amount, Hourly Rate + Flat Amount, Hourly Rate, Percent, and Points.

Rate Combinations

The combination of rate types and conversion rates with account types that is linked to your budgeting model. Typical rate types are current, commercial, floating, average, and historical. Effective dates define different rates for different budget periods. There are several conversion rates for any pair of currencies including not only the current rate, but others rates such as average, historical, asking, and so on. These different types of rates are appropriate for different types of accounts.

Rates

The arrays of values used to calculate the cost of a plan to an employee. Rates can be age-graded, service-related, or general, depending upon the benefit plan type. Any number of benefit program and benefit plan combinations can use each set of rates.

In Enterprise Performance Management, a rate is determined by the user and specifies the dollar amounts to be calculated for each model. This is a financial services industry term.

Rating Model

The scale used by your company to measure competency proficiency. The default rating model is the PSCM (PeopleSoft Competency Management) Rating Model that PeopleSoft delivers with your PeopleSoft Human Resources System.

Raw Punches

See Actual Punch; typically this is distinguished from a rounded punch (see Time Reporting)

Reason Code

Reason Codes provide explanations for occurrences such as returned stock and changes to order headers, lines, or schedules.

Reason Code

A code describing employee time such as comments for sick time or travel time.

Reasonableness

In PeopleSoft Demand Planning, a technique that checks the trend and projected annual growth to make sure that a forecast is realistic. If a forecast falls outside either boundary, the system automatically adjusts it and sends a warning message.

Reassignment

Change of an employee from one position to another without promotion or change to lower grade.

Recalculate Forecast

In PeopleSoft Demand Planning, a forecasting feature that uses the existing model and its associated parameters to create a new forecast.

Recalculate VAT at Payment

Allows the VAT amount to be adjusted at the time of payment if an early payment discount is taken. This calculation option is only valid when VAT is calculated at Gross.

Receipt Cost Method

Determines how you cost receipts. Receipt cost methods include Actual, Non-Cost, and Standard.

Receipt Line

A line associated with a Receipt ID that identifies an item and quantity. If the respective tracking is activated, the lot, lot suffix, and serial number are also identified.

Receivable Update

See **Posting**

Receivables Item

An individual receivable. An item can be an invoice, credit memo, or debit memo. Items and payments combined comprise a customer's balance.

Reconciliation

Within PeopleSoft Enterprise Performance Management, reconciliation differs slightly when it is performed within the PeopleSoft Enterprise Warehouse and when it is performed within the PeopleSoft Analytic Applications.

In PeopleSoft Funds Transfer Pricing (FTP) and PeopleSoft Risk Weighted Capital (RWC), reconciliation identifies differences between Performance Ledger balances and the instrument or position balances, which are risk weighted according to the basis rules you have assigned. The first step in reconciling basis rule balances is to reconcile the individual balances for accounts, instruments, and positions. Reconciling the total balances is the second step. This means that you reconcile the difference between Account/Instrument balances, and the difference between Account/Position balances.

In the PeopleSoft Enterprise Warehouse, reconciliation is a period-end process that posts journal entries to the Performance Ledger for the discrepancies found when you reconciled the individual balances. Typically, you'll run the PF Reconciliation engine after a period to compare "to and from amounts" between tables such as REVENUE_F00 and PF_LEDGER, or the GL_LEDGER and the PF-LEDGER.

Record Date

The date a stockholder must officially own shares in order to vote at the meeting or to derive an adjustment resulting from a stock split or a stock dividend. The Board of Directors sets the Record Date.

Record Definition

A logical grouping of data elements.

Record field

Once a field is inserted into a record definition it becomes known as a Record Field within the record.

Record Group

A set of logically and functionally related control tables and views. Record groups exist for two basic reasons:

- To save you time—with Record Groups, TableSet sharing can be accomplished quickly and easily, eliminating an enormous amount of redundant data entry
- To act as a safety net—Record Groups ensure that that TableSet sharing is applied consistently across all related tables and views in your system.

Record Input VAT 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 being tracked on a transaction, this flag is always set to Yes. This flag is not used in Order Management, Billing, or Receivables, where it is assumed that you are always recording only output VAT, or in Employee Expenses, where it is assumed that you are always recording only input VAT.

Record Output VAT Flag

For certain transactions within PeopleSoft Purchasing, Payables, and General Ledger, it may be necessary to record both input VAT and output VAT on the same transaction. Generally, this would be a situation where the purchaser was required to self-assess VAT. Within these situations, this flag indicates that you are recording output VAT on the transaction. This flag, in conjunction with the Record Input 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. This flag is not used in Order Management, Billing, or Receivables, where it is assumed that you are always recording only output VAT, or in Employee Expenses, where it is assumed that you are always recording only input VAT.

Record Owner

The "Stockholder of Record" of the stock. This may be different from the "Beneficial Owner" of the stock.

Record Suites

Record suites are temporary tables that enable the system to track how many processes are running. These temporary tables leave the fact tables accessible for processing other jobs simultaneously without impacting your processing.

Reduction In Force (RIF)

Method used to reduce the number of government workers in an agency.

Reemployed Annuitant

An employee who has retired from Federal employment and is receiving an annuity. His/her salary is reduced by the amount of the annuity.

Reference Designators

A user-defined alphanumeric identifier that determines where a component is placed in an assembly.

Reference Transaction

In People Soft 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.

Referential Integrity

Issues that occur when an update to an instance of one object invalidates one or more instances in a related object. In other words, when you make a change to one area of the application, referential integrity makes sure the changes do not adversely affect another area of the application.

Refresh Time

The process that retrieves the appropriate current version of objects related to employee time (such as task profiles or work schedules) and associates them with that time.

Region Codes

Regions may or may not be physical entities, they may simply be another way to geographically categorize an area. When a region does represent a physical entity, the region code has the same characteristics as a business, that is, an address and a language spoken.

Register of Separations and Transfers (ROST)

The ROST is a regulatory compliance document used by federal agencies to summarize the information in an employee's Individual Retirement Record (IRR). The ROST is a one-page cover sheet that accompanies a batch of IRRs being submitted to the Office of Personnel Management (OPM) at the time of an employee's separation from a federal agency. Employees covered by the CSRS retirement plan require SF-2807. Employees covered by the FERS retirement plan require SF-3103.

Registration

The name or names that appear on the stock certificate to indicate who owns the stock.

Registration Statement

The document that must be filed to permit registration of an issue of securities under the Securities Act of 1933. A major component of the registration statement is the prospectus that is to be supplied to prospective purchasers of securities.

Regression Analysis

A statistical technique that determines the relationship between two or more variables. Regression predicts the value of one variable (the dependent variable) based on one or more independent variables.

Regular Base Compensation

In PeopleSoft Workforce Analytics, the annualized, quoted, compensation rate for a job. Consists of fixed compensation, does not include variable compensation.

Regular Time

An employee's normal (scheduled/shift) work hours.

Regular Time

In PeopleSoft Workforce Analytics, an employee's normal (scheduled/shift) work hours.

Regulation T

Federal Reserve Board regulations governing the extension of credit by brokers or dealers, including their participation in same-day sale transactions and sell to cover exercise.

Regulatory Region

The Regulatory Region functionality in PeopleSoft HRMS is designed for use in performing regulatory and regional edits. You'll use Regulatory Region to drive PeopleCode edits, perform set processing, and control what codes and values the operator sees. A Regulatory Region can be any country (or province or state) where there are specific laws and regulations addressed by functionality in PeopleSoft HRMS.

In Enterprise Performance Management, a Regulatory Region is a region with a common regulatory framework; such as a country (CAN for Canada), or a smaller state or provincial entity (CANBC for British Columbia).

Related Education

Education above the high school level that has equipped the applicant with the KSAs to perform successfully the duties of the position being filled. Education may relate to the duties of a specific position of to the occupation, but must be appropriate for the position being filled.

Release

An industry standard term associated with the lifting of a company's Repurchase Option from a portion or all shares from a Restricted Stock Award (RSA). RSA's are subject to release schedules, similar to vesting schedules.

Relevant Constraint

A constraint PeopleSoft Planning considers when it calculates a score and when it optimizes the schedule. See also **Scorecard** and **Optimize**.

Reloads

Some stock option plans provide for the grant of a "reload" stock option in connection with stock option exercises, typically by means of stock swaps. A reload option feature provides that upon a stock exercise, the employee will receive an automatic grant of a new stock option at the then-current fair market for the shares that they exercised or for the shares that they used to swap.

Remark Codes

Codes that cause the printing of pre-set text passages on a notice of action form. Some passages are general purpose and others are specific to the personnel action being processed.

Remit From Customer

A customer who is responsible for payments billed to other customers. During cash application, it's useful to look at open items for the Remit From group.

Remittance Worksheet

A work space in PeopleSoft Receivables used to select drafts for remittance to the bank.

Reorder Point

The identifier that automatically locates a replenishment need for an inventory item. When the physical quantity in a location falls below the reorder point, a replenishment request can be created.

Reorder Point Policy

In PeopleSoft Inventory Planning, a policy that determines when a replenishment order is launched for an item. The policy has several methods that include days supply, lead time demand, and **Fixed Quantity**.

Replacement Option

The "new" "replacement" stock option that will replace the original stock option. This option will have a grant price lower than the original stock option.

Replenish

A process that indicates when items need to be resupplied from external sources. In PeopleSoft Inventory, the process can occur on an ad hoc basis or at predefined reorder points.

Replenishment Request

In PeopleSoft Flow Production, an online request for material made when the material is needed. You can generate replenishment requests manually or automatically using backflushing. You can communicate that request using pull lists, pull tickets, or Workflow notifications.

Report Scope

A feature that creates multiple instances of an nVision report using a single report request. Each instance contains data specific to an individual ChartField, such as a business unit or department, or for a group of ChartFields, such as all sales departments. Using Scope, each report instance can share the same layout while containing data unique to these particular ChartFields.

Reported Time

Clock time or elapsed time provided to the system by the user (see Time Reporting)

Reporting Person

An insider that is regularly considered by the SEC to have material information and policy-making authority for the corporation. These individuals are subject to the reporting requirements promulgated by Section 16 of the Securities Exchange Act of 1934. Reporting Persons typically include Directors, Officers, and shareholders with 10% holding interest in the equity of the registrant's securities.

Repricing

An agreement between the corporation and the optionee that allows the optionee to cancel an outstanding high-priced, usually "Out-of-the-Money" stock options for lower-priced options.

Repricing Election

Eligible optionees can choose (elect) to accept the corporation's repricing offer or choose to decline the offer.

Repurchase

The reacquisition of shares of stock from an individual by a corporation. This usually occurs when an individual fails to meet the vesting requirements on a RSA or option that is exercised before it vested. The corporation might pay the original cost of the shares to the individual or the fair market value of the shares at the time of repurchase.

Repurchase Option

An irrevocable, exclusive option to repurchase up to the number of shares that constitute Unreleased Shares at the original purchase price per share. The Company shall exercise said option. The repurchase of outstanding shares is regulated under the laws of all states (except Massachusetts). Under some laws, as under the Model Business Corporation Act, the repurchase is prohibited unless the corporation remains solvent, in both the equitable and bankruptcy senses of insolvency and after taking any liquidation preferences of other outstanding stock into account.

Repurchase Right

A company's contractual right to buy back from an employee any stock resulting from the exercise of the option. The buy back can be at fair market value, book value, or the original purchase price.

Reservation Method

The method used to reserve soft reservation items — either batch COBOL reservations or on-line allocation and reservation.

Reserved

A flag indicating that the inventory item is reserved for stock fulfillment in the inventory business unit.

Reserved Orders

Orders that have been reserved against on-hand available quantity at the business unit-item level. Reserved orders are found in the DEMAND_INV table.

Reset

In PeopleSoft Demand Planning, a function of the **Forecast Calculation Process** that determines which forecast model will produce the best forecast, meaning the model with the lowest ratio of error.

Resolution

An activity that closes or partially closes a deduction, such as matching it to a deduction authorization, writing it off, or sending it back to PeopleSoft Receivables.

Resolution Entry Type

Code that identifies how to process activities for items in PeopleSoft Deduction Management and how to create accounting entries.

Resolution Method

A set of rules that defines how to automatically match or write-off deductions in PeopleSoft Deduction Management.

Resolution Worksheet

The workspace in which deduction items are paired with offset items and resolved or written off in PeopleSoft Deduction Management.

Resource

In PeopleSoft Manufacturing, any crews, machines, and tools that can optionally be used at work centers to complete tasks. In PeopleSoft Performance Measurement, any homogeneous grouping of general ledger line items.

Resources

Resources are the economic elements that are required to perform the activities associated with your business. Resources are consumed in the performance of these activities, and thus denote operating costs. In PeopleSoft Activity-Based Management, resources are typically regarded as the groupings of one or more general ledger accounts. In a service business, resources might include salaries, office rentals, and costs of capital such as information systems, depreciation, real estate taxes, and other associated costs.

Resource Amount

The monetary amount of a single, specific resource transaction. The Resource Amount maps to the Posted Total Amount when posted to the general ledger.

Resource Category

A field for defining individual resource types more specifically. For example, if you have a resource type of labor but want to break it down further for tracking purposes, you might define resource categories of architect hours, carpenter hours, plumber hours, and electrician hours. Resource categories are optional.

Resource Driver

In Activity-Based Management, a Resource Driver defines the quantity of resources used by an activity.

Resource Group

A category of resource types. You can define relationships between the resource types within a resource group to facilitate analysis of project costs. For example, if you had resource types for standard labor and overtime labor, you could group them together in a resource group to calculate total labor.

Resource Planning

In PeopleSoft Activity-Based Management, Resource Planning focuses on resources allocations that create expected results like driver rates and cost object costs.

Resource Quantity

A field on each resource transaction that identifies nonmonetary quantity. For example, on a resource line for 12 ball valves the quantity would be 12.

Resource Source

A field on each resource transaction that identifies the system in which the cost originated. For example, PeopleSoft Payables would be the resource source for a resource transaction created from a voucher in that system.

Resource Subcategory

A field for defining individual resource types and categories more specifically. For example, if you have a resource type of labor, and resource categories of architect hours, carpenter hours, and plumber hours, you might want resource subcategories of regular hours and overtime hours. Resource subcategories are optional.

Resource Supplied

An attribute that enables you define a resource as committed or flexible. A committed resource is one that will not likely change in the short term. A flexible resource is more likely to change within the short term.

Resource Transaction

An individual cost line within PeopleSoft Projects. It is through resource transactions that individual costs and types of costs are tracked. Each resource transaction contains a cost and a quantity and as many identifiers of that cost as necessary. Resource transactions are created when you receive information from other systems, run allocations with project resources as the target, or perform internal transactions such as billing, project closure, or adjustments.

Resource Type

The resource transaction field in PeopleSoft Projects that identifies the resource associated with a given cost. Resource types may be very general or very specific depending on your needs; they are used in conjunction with resource categories, resource subcategories, and resource groups.

Resource Use

Resource Use defines the behavior of a resource within PeopleSoft Enterprise Performance Management. An intermediate resource is a grouping of general ledger line items that may be allocated to another intermediate resource or to a primary resource.

Restricted Punch

A punch which is not accepted because it occurs outside of the predefined number of hours and minutes before or after a scheduled (Understanding Time Collecting Device)

Restricted Securities

Securities issued privately by the company, without the benefit of a registration statement. Restricted securities are subject to a holding period before they can be sold under Rule 144.

Restricted Stock Awards (RSA)

An award of shares of stock to an individual, typically granted at the par value or for no consideration. The shares are awarded on the basis of some future performance goal, either the passage of time (vesting) or the attainment of a specific goal. When the goal is achieved, the vesting occurs. The individual, typically, has all other shareholder rights over these shares such as, voting and dividend rights. The shares are issued in the name of the individual at the time of the award and are held in escrow until vesting occurs. If an employee terminates prior to the vesting of the shares then the company normally repurchases the unvested shares.

Retained Grade Effective Date

Date employee became eligible or began receiving a retained grade and pay.

Retained Grade Expiration Date

Expiration date of an employee's retained grade and pay.

Retest Date

In PeopleSoft Inventory, the date a lot should be inspected to determine whether it is still acceptable for fulfillment or consumption. (Retest Date = Creation Date + Retest Lead Time)

Retirement

Types of retirement are:

- Mandatory Retirement.
- Disability Retirement.
- Voluntary Retirement.
- Special Option Retirement.
- ILIA (In Lieu of Involuntary Action) Retirement.

Retirement Coverage Code

A code used to denote an employee's retirement coverage. The major ones include the following:

- Civil Service (CSRS)
- Federal Employees Retirement System (FERS) and FICA
- Foreign Service (FS)

- CSRS Offset
- CSRS - Special (for LEOs)
- FERS and FICA - Special (for LEOs)
- Social Security System
- None

Retroactive Benefits/Deductions

Deductions taken or benefits granted due to a recalculation of previous benefits and deductions. Late or modified union contracts, late paperwork, and delays in benefit enrollment processing may all result in a need for benefit/deduction recalculation.

Return Type Code

A designator on returned material authorizations (RMAs) that indicates what actions the return initiates. This may include replacement of the product or the creation of a credit memo in PeopleSoft Billing.

Reverse Split

A reduction in the number of outstanding shares of a corporation's stock, with a corresponding increase in the stock's value.

Reversionary Annuity

A form of pension payment where the retiree foregoes all benefit during his or her lifetime so that the entire benefit is paid as an annuity to a beneficiary after the retiree's death. If the beneficiary predeceases the retiree, the benefit is forfeited.

RIDDOR (Reporting of Injuries, Diseases, and Dangerous Occurrences Regulations)

Health and safety regulations in the United Kingdom requiring employers to report certain types of health and safety incidents to the Health and Safety Executive (HSE).

Rider

A special court-ordered or regulatory provision that may be applied to an enrollment to expand or limit any dependent or beneficiary coverage.

Risk Function

In Risk Weighted Capital, this is a user-defined formula that the system uses to derive risk weights.

Risk RuleSet

Used to assign a number of rules to a basis, for processing by the PeopleSoft Risk Weighted Capital Application. Used to group together a number of rules that frequently apply to the same type of balance.

Risk Type

In Risk Weighted Capital, this defines the types of risk associated with your business or activity. For example catastrophic, credit, legal, operational, regulatory, foreign exchange, market, interest rate.

Risk Weight

In Risk Weighted Capital, the risk weight is assigned by risk type, and is used to calculate capital allocation or normalized loss for the account or activity.

Risk Weighted Capital (RWC)

See PeopleSoft Risk Weighted Capital

RIZIV Code

This code is for Belgian employers to track the Federal Institute for Illness and Disability Insurance category.

ROE (Record of Employment) Reason Codes

ROE codes are defined by the Canadian government for employers to record employment actions such as Return to School or Pregnancy Leave.

ROLAP (Relational Online Analytical Processing)

ROLAP refers to the analytical processing and analysis of a relational Data Mart cube. ROLAP, is a form of OLAP that leverages the power and flexibility of relational databases.

Role

A role consists of a designated set of tasks, competencies and accomplishments required for a job code or a position.

Role user

A PeopleSoft Workflow user. A person's Role User ID serves much the same purpose as their Operator ID does in other parts of the system. It allows the system to uniquely identify the user and to determine what data the user has access to.

PeopleSoft Workflow uses Role User IDs rather than Operator IDs because it needs different kinds of user information than the rest of the system does. Specifically, it needs to know how to route work items to the user---an email address, for example---and what role the user plays in the workflow. Plus, you can include role users in your workflow who aren't PeopleSoft application users and who don't have Operator IDs.

Roles

Roles describe how people fit into the 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 Forward

In commitment control, rolling budget balances forward from the budget ledger you are closing (the source budget ledger) into the new (target) budget ledger.

In PeopleSoft Enterprise Planning and Production Planning, a utility that moves tasks from the past to a valid point in the future using time periods rather than fixed dates.

Roll Up

The act of totaling sums based on information tree hierarchies. You can roll up data for any group of details that you have defined as dependent with the Tree Manager.

Rolling Plan

An ESPP offering period where the purchase date is measured from the offer start date. If at the purchase date, the current stock price is lower than the last stock purchase price, you may elect to reset your employees to the new lower purchase price. The offering period is now based of the new purchase date.

Rollup

In PeopleSoft Demand Planning, the process of adding up the demand and forecast **Array** from one level to the next from child to parent. Information such as caption, description, and unit of measure can also be rolled up. The process is also referred to as summarization.

Rounded Punch

A punch that has a company's rounding requirements applied to it (see Time Administration)

Rounding Rule

Defines a rounding rule. You use rounding rules in numerous places—for instance you could round an earning, deduction, or many of the elements that make up an earning or deduction.

Routing (Manufacturing and Engineering)

A set of information detailing the method to manufacture a particular item. It consists of sequentially numbered operations that reference the task to be performed, the work center in which the task is to be performed, the resources to be used, and the time required to complete the task. Engineering Routings differ from Manufacturing Routings in that they are not visible within Production Planning or Production Management and are isolated from Manufacturing.

Routing Option

In PeopleSoft Planning, a valid method for replenishing supply for an item. There are two types of routing options: build options and purchase options. An item may have more than one routing option.

Routing Transit Number (RTN)

A number that identifies the financial institution to which an electronic payment should be sent for deposit.

Routings

Routings connect the activities in the workflow. They are the system's means of moving information from one place to another, from one step to the next. Routings specify where the information goes and what form it takes—email message, electronic form, or worklist entry.

RSZ (Rijksdienst Sociale Zekerheid) Category Codes

These government defined Social Insurance category codes are used to maintain social security records for your Belgian employees. RSZ Categories are associated with a Contract Type, Statute and Substitute for Claeys Formula calculations.

Rule

Representation of a company’s compensation, task allocation, or exception requirements (see Creating Rule)

Rule 10b-5

A SEC rule that prohibits trading by insiders on material non-public information. This is also the rule under which a company may be sued for false or misleading disclosure.

Rule 144

A SEC rule that applies to public re-sales of restricted securities as well as all sales by affiliates. The requirements include (1) current public information about the issuer, (2) a one-year holding period for "Restricted Securities," (3) unsolicited brokers' transactions, (4) an amount limitation. the greater of 1% of the outstanding stock or the average weekly trading volume may be sold during any three-month period, and (5) a Form 144 filing.

Rule Actions

Functions that can be used in the creation and application of a rule (see Time Administration)

Rule Elements

Customer defined pieces of information which are passed to Time Administration in order to apply and evaluate rules (see Time Administration)

Rule Period

A Time & Labor period used in the evaluation and application of a rule (see Time Administration).

Rule Program

Specifies the set of rules the Time Administration process will execute and the order in which it will execute the rules.

Rule Results

Net effect of the application of a rule; for instance, the creation of time, initiation of workflow, modification of reported tasks (see Time Administration)

Rule Templates

Templates used to quickly create a variety of rules for the Time Administration program to execute when processing reported and/or scheduled time. Some examples are. compensation rules for overtime and holidays, notification rules for irregular attendance, and rules for just about any other time-reporting situation that requires special processing.

Rules

Rules are your company's business practices captured in software. Rules determine what activities are required to process your business data.

Rule Set

Rule Sets enable you to apply basis rules to your PeopleSoft Analytic Application in the sequence that you prefer. This is particularly helpful if there are multiple basis rules for the same account node, product node, or position source code. Rule Sets can control the execution sequence of your rule combinations, filter combinations, or both. The first occurrence on the node will be applied and any other occurrence will be ignored. Rule Sets are also used with the Data Manager, and with the Currency Conversion engine.

Rules/Time Administration

A physical implementation or execution of a company’s compensation, exception and task rules (see Time Administration)

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 some type of program that manipulates data in some way.

Run Control ID

A unique ID to associate each operator with their own run control table entries. Process Scheduler.

Run ID

Code that uniquely identifies a Run Control for batch processes.

RWC (Risk Weighted Capital)

See PeopleSoft Risk Weighted Capital

S

Safety Stock Policy

In PeopleSoft Inventory Planning, a policy that determines how safety stock quantities are calculated for an item. The policy has several methods that include days supply and percentage of demand fill.

Salary

Rate of compensation received by an employee.

Salary Group

Part of a group of defaults assigned to job codes. A salary group may include items such as steps and grades dependent on individual company parameters.

Salary Plan

A plan of salary defaults, grades, and step components

Salary Step Components

Pay components assigned to a salary step by entering the corresponding rate codes on the Salary Step Components page.

Salvage Value

An estimate of the amount of money one might receive upon selling an asset once that asset reaches its useful life. Salvage value is used in several depreciation calculations, including Straight Line.

Sales Order Rebate and Penalties

Rebates or penalties that are calculated against sales orders independent of **Buying Agreement**.

Sales Person

A required field used in PeopleSoft Receivables, Billing, Order Management, and Deduction Management when working with items. Each item must be assigned to a credit analyst. If no

credit analyst is assigned to an item, the credit analyst assigned to the customer is used as the default.

Sales UOM

The only units of measure that can be referenced on sales orders and quotes. You establish them on the Product Attributes by UOM page.

Same-Day Sale

An exercise and sale occurring on the same day. The exercise of the option and sale of the underlying shares take place simultaneously. The broker uses the proceeds of the sale to pay the company the exercise price and any tax withholding and the optionee the net cash (less any brokerage commission/fees).

Sample Method

One method of entering characteristic readings for a quality control plan. Using this method, for one given control plan you inspect all the characteristics for the first sample, then all the characteristics for the next sample and so on.

Scale

On a Goals Matrix (In PeopleSoft Workforce Analytics), a scale that defines the lowest, middle, and highest levels of performance needed to achieve associated minimum, midpoint, and maximum levels of compensation pay out. These are referred to as the Threshold, Target, and Stretch levels, respectively. The scale can be used to standardize multiple performance goals to a common scale of measurement.

Scenario

A scenario is a particular outcome you are analyzing when you run in Scenario Manager. Scenarios enable you to study various changes in organization models you created. For each parent and child model you want to study, you create a scenario ID that you use with all run controls.

There are two types of scenarios defined in the Scenario Manager: Historical and Forecast. In the case of an Historical Scenario all future looking fields will be display only and the Scenario Manager component serves as a wrapper to run any analytic engines. In the case of a Forecast Scenario, the Scenario Manager refers to all the business rules, forecasts, and economic assumptions that make up the scenario.

In PeopleSoft Budgeting a scenario is a ChartField used in PeopleSoft Budgeting to identify different budget iterations that use different assumptions.

Schedule

Specific task, date, and time to be worked by a Time Reporter (see Scheduling)

Schedule 13D or 13G

Disclosure forms required to be filed with the SEC and the company by a shareholder (or shareholders) that own(s) more than 5% of a public company. Schedule 13G is a short-form version of the 13D and may generally (but not always) be used only by institutional investors.

Schedule Group

A category of employees or employee groups associated for purposes of time scheduling.

Schedule Line Number

The line associated with an Order ID. The schedule line identifies an item and scheduled ship quantity that may be different from the requested quantity due to item availability.

Schedule Number

A number identifying the salary table form that an employee's pay is computed. Also has a second meaning related to the Payment Voucher processing for the ECS.

Schedule Reconciliation

In PeopleSoft Payables, the process of reconciling scheduled payments by Payment Schedule ID. Schedule Reconciliation helps U.S. federal agencies meet their requirement to schedule or group together payment orders for submission to the Treasury Disbursing Office.

Schedule Template

An ordered pattern of workday(s) and/or off day(s) used in scheduling (see Scheduling)

Schedule Type

In PeopleSoft Payables, an indicator of the nature of items purchased with a Payment Schedule**.**

Scheduled Punch.

A time reporter’s expected punch (see Scheduling)

Scheduling

A function of PeopleSoft Time and Labor and PeopleSoft Global Payroll that enables you to create work schedules and assign them to employees.

Scorecard

A weighted sum of constraint violations in a schedule that evaluates the schedule's validity (that is, acceptability). The score is calculated by adding the value for each relevant violated constraint. See also **Penalty, Weight** and **Relevant Constraint.**

In PeopleSoft Balanced Scorecard, views of a strategy tree’s components and Key Performance Indicators with red, yellow, or green scores that show its assessments.

Scrap

Any material outside of specifications and possessing characteristics that make rework impractical.

SearchIndex

A set of objects that give the programmer the ability to create, delete, insert, and update a search index and the items within it. Search index items contain a set of statistics about the document that has been indexed (keywords, number of occurrences, proximity to other words, and so on) as well as a key that can be used to point to the document (a URL, database key, or file path).

SearchQuery

A set of objects that allow the programmer to pass a query string and operators to the search engine and receive a set of matching results with keys to the source documents from the search index in return.

Seasonal Index

In PeopleSoft Demand Planning, measures the amount by which a forecasting period is influenced by seasonal effects. The index typically measures the percentage of difference between the base demand in the period and the expected actual demand in the period. An index of 100 indicates an average period in a seasonal cycle.

Seasonal Profile

In PeopleSoft Demand Planning,identifies the weight index assigned to a forecast time period to take in account seasonal fluctuations in the demand.

Seasonality Group

In PeopleSoft Demand Planning, a group of **Forecast Items** with a repetitive pattern of demand from year to year where some periods are higher than others. Typically a group of items is designated as a contributor to the seasonality group. Contributors are chosen because they are representative of the group, are stable, and have at least two years of demand history. The seasonality group profile is more stable than individual profiles of the contributors because the Aggregation process smoothes out random errors.

In Enterprise Planning and Simulation, a Seasonality group is a group of items with similar seasonal patterns. To determine if a forecast element is seasonal or nonseasonal, by averaging their history values over a year and determining where they were above and below average. Seasonality groups capture means seasonal behavior among related products.

Secondary COBRA Events

COBRA qualifying events that extend the amount of time a participant is eligible for COBRA coverage. For an event to qualify as a secondary COBRA event, it must fulfill the following qualifications: The participant must already be enrolled in COBRA coverage as a result of an initial COBRA event, the initial COBRA event must be one that is associated with a change to the employee's job status (such as a reduction in hours, termination, or retirement), and the

secondary event must be one of the COBRA event classifications that involves loss of coverage for the dependent (such as divorce, marriage of dependent, or death of employee). See COBRA and Initial COBRA Events.

Secondary event code

Secondary event codes, also called transaction codes, specify the type of transaction in detail. For example, a transaction's secondary event code could say that the transaction is a catalog order, a rush order, or a request for a sample. Not all transaction types include secondary event codes.

Section

A set of logically related elements that are to be resolved during the payroll process. You define your payroll process by creating sections and adding them to process lists. You can create four different types of sections: standard, generate positive input, sub-process, and payee.

Section 16(a)

Provision of the Securities Exchange Act of 1934 that requires company insiders to file changes in beneficial ownership of the company’s equity securities and periodic reports disclosing their holdings.

Section 16(b)

Provision of the Securities Exchange Act of 1934 that requires that any profit realized by a company insider from the purchase and sale, or sale and purchase, of the company’s equity securities within a period of less than six months must be returned to the company. It is also known as the "short-swing profit" rule.

Section 423

The Internal Revenue Code section that regulates Employee Stock Purchase Plans.

Section 83(b) Election

A tax filing within 30 days of grant that allows employees granted restricted stock to pay taxes on the exercise date, rather than the date when restrictions lapse. If an employee files the election, taxes are based on the fair market value on the exercise date, with any future appreciation taxed as a capital gain. If the employee does not file an election, taxes are based on the fair market value on the date the restrictions lapse, which will be higher assuming the stock has appreciated in value.

Securities Act of 1933

Often referred to as the "truth in securities" law, the act requires that investors receive financial and other significant information concerning securities being offered for public sale; and prohibits deceit, misrepresentations, and other fraud in the sale of securities.

Securities Exchange Act of 1934

The Congressional act that created the Securities and Exchange Commission. The Act empowers the SEC with broad authority over all aspects of the securities industry. This includes the power to register, regulate, and oversee brokerage firms, transfer agents, and clearing agencies as well as the nation's securities self regulatory organizations (SROs). The various stock exchanges, such as the New York Stock Exchange, and American Stock Exchange are SROs. The National Association of Securities Dealers, which operates the NASDAQ system, is also an SRO. The Act also identifies and prohibits certain types of conduct in the markets and provides the Commission with disciplinary powers over regulated entities and persons associated with them. The Act also empowers the SEC to require periodic reporting of information by companies with publicly traded securities.

Security Clearance

Security Clearances (Classified, Secret, Top Secret) are granted to employees by government agencies and are usually associated with jobs that bring employees into contact with classified government projects or sensitive technologies.

Security Event

In commitment control, events that trigger security authorization checking, such as budget entries, transfers, and adjustments; exception overrides and notifications; and inquiries.

Segmentation

You can “segment” components of pay based on such events as changes in compensation, employee status, or job changes during a pay period. For example, if an individual changes jobs in the middle of a pay period and your organization has a practice of separating components earned in the first job from those earned in the second job, you can set up your system to trigger segmentation of earnings results on the pay slip when there is a change to the job change action/reason field.

Selective Factors

Knowledge, skills, abilities or special qualifications that are in addition to the minimum requirements in a qualification standard, but are determined to be essential to perform the duties and responsibilities of a particular position. Applicants who do not meet a selective factor are ineligible for further consideration.

Selective Merge

In PeopleSoft Enterprise Warehouse the selective merge allows for an additional WHERE clause when you delete a merge.

Self-Service Application

Another name for PeopleSoft's HRMS and ERP applications accessed using a browser.

Sell to Cover Exercise

When an optionee sells a portion of the option shares to cover the exercise cost and any applicable taxes.

Seniority Pay

A premium paid for seniority or for the length of time an employee works for an organization.

Seniority Pay Components

Pay components whose rate codes are assigned to the seniority rate code class SENPAY (provided by PeopleSoft) on the Comp Rate Code page, allowing you to access the seniority pay functionality.

Seniority Rate Codes

A premium paid for seniority or for the length of time an employee works for an organization.

Separate Debit and Credit

A feature in PeopleSoft General Ledger that captures and reports in greater detail the accounting information that resides in balance sheet accounts. It shows the gross debit/credit balances in addition to the net balance for each account stored in the ledger. This feature also supports reversing—debit and reversing—credit journal entries for error correction.

Server Process

A server process is executable code that receives incoming client requests on the application server. The server process carries out a client request by making calls to a service that executes SQL against the database.

Service

A service performs a particular task of an application. Examples of services are MgrGetObj, SQLAccess, RemoteCall, and so on. When a client workstation sends a request to the application server, it sends a service name and a set of parameters, such as "MgrGetObject + parameters". The application server associates the service request with the appropriate server process to complete the transaction.

Service

The PeopleSoft Pension Administration function that determines how much service credit an employee has accrued.

Service Buy Back

The process by which an employee repays a pension plan in order to restore service credit that was forfeited when the employee withdrawal previous contributions. Typically, employees withdraw contributions upon termination and initiate service buy back processing upon rehire.

Service Purchase

The process by which an employee gets additional pension service credit for periods not normally considered eligible. The employee “purchases” this service by paying into the plan.

Service Schedule

A table showing how much service an employee earns based on the number of hours the employee worked during the year or month.

SetID

The label that identifies a TableSet.

Sex Code

Used to indicate gender.

Shape

For a transaction, the set of inputs and outputs for that transaction. For a class, the data members of that class.

Share

A share of a company's stock. Stock options give you the option to purchase a certain number of shares of company stock.

Share Price

The price per share of a company's stock. See, also, "stock price."

Shareholder

Owner of one or more shares of stock in a corporation. Also known as a stockholder or investor.

Shares Available to Issue

The total number of shares authorized, less shares granted, plus cancellations that revert to the Plan pool.

Shares Cancelled

This is usually triggered by a specific event, such as termination of employment in which the unvested shares as of the date of termination are no longer available for future vesting and exercise. These shares are therefore canceled from the option and can be returned to the plan, retired to treasury or allocated back to a group.

Shares Exercisable

The number of shares that are vested and available for exercise.

Shares Exercised

The number of shares purchased upon exercise of a stock option.

Shares Expired

Option shares that no longer are exercisable at the end of the option term. The length of the option term is defined in option agreement. This date is usually the earlier of the exercise period for vested shares after termination of employment or the full length of the option term.

Shares Outstanding

The number of company shares currently held by shareholders, as tracked by the transfer agent

Shift [Time and Labor]

The block of hours that an employee works in a day, such as nine to five, four to eleven, or ten to six. In PeopleSoft Time and Labor, Shift is used as a template of clock hours for scheduling an employee or group of employees to be at work or available to work (on call). Shifts may be constant, rotating, repeating, and/or split; any given shift may or may not have an associated Shift Differential or Bonus. A shift is always associated with a Work Schedule, and consists of clock hour Start and Stop times (two to allow for split shifts), meal periods (two) and relief periods (two).

In PeopleSoft Workforce Analytics, the block of hours that an employee works in a day such as nine to five, four to eleven, or ten to six.

Shift Bonus

A fixed amount (either a flat dollar figure or stated in terms of an employee's rate) paid for working a particular Shift.

Shift Code

A numerical shift identifier that is unique within a SetID.

Shift Differential

Additional compensation paid an employee for time worked during certain shifts. Typically, shift differential is administered as a flat amount per shift, hour worked, and/or as a percentage of the amount paid per shift hour or shift worked.

In PeopleSoft Workforce Analytics, a premium paid for work over regular pay, for which employees on certain shifts may be eligible, such as double-time for late night shifts. Shift differentials are usually stated as an additional rate or factor.

Shift Elements

Individual components of a shift such as TRC start and stop time, duration (see Scheduling)

Shift Name

Customer defined nomenclature for a shift (see Scheduling)

Shift Type

A customer-defined classification associated with a shift. The shift type can be used in the evaluation of rules or exceptions (i.e. On Call) (see Scheduling)

Shift Type [Time and Labor]

Time and Labor defined classification of shifts. Valid shift categories include Flex, General and Elapsed. Shift categories are used in the creation of time reporter schedules (see Scheduling)

Shipping Priority Code

Shipping Priority Codes act as tie breakers during order reservation in PeopleSoft Inventory when different orders are scheduled for the same delivery date and time. When the reservation process in Inventory encounters a situation where there are more orders than available stock, the system reserves the order with the highest priority. If schedules are encountered with the same priority, orders are then considered by order date.

Short Sale

The sale of a security that is not owned or is not delivered at the time of the trade, necessitating its purchase or delivery some time in the future to "cover" the sale. A short sale is usually made with the expectation that the stock value will decline, so that the short seller can eventually cover at a price lower than the original sale, thus realizing a profit. At the time of the short sale, the broker borrows stock to deliver on the settlement date. A short sale can be "naked," in which case the seller does not deliver the shares being sold short and must provide the broker with collateral. Or the short sale can be "against the box," in which case the seller delivers the shares being sold short for the broker to hold "in the box" until the seller chooses to close out the short position.

Short-Swing Transaction

A purchase and sale, or sale and purchase, of the issuer's equity securities by an insider within a period of less than six months. See "Section 16(b)" above.

Short-term Variable Compensation

In PeopleSoft Workforce Analytics, this is cash compensation paid to a worker that is not fixed; includes bonuses and commissions.

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.

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.

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.

Sick Leave

Sick leave is accrued by full-time permanent/seasonal employees at the rate of 4 hours every biweekly pay period; for part-time permanent/seasonal employees, it is accrued at one hour for every 20 hours worked.

Simulated Workforce

In the PeopleSoft Workforce Rewards product, Manage Compensation module, the calculated Simulated Workforce = Existing Employees + New Simulated Employees + Reduced-Employees.

Single Life Annuity

A benefit payable during the lifetime of the participant, with no payments made after the death of the participant. Also referred to as a “life only annuity” or a “straight life annuity.”

Single Signon

This refers to the process by which a user can, after being authenticated by one PeopleSoft application server, access a second PeopleSoft application server without entering a user ID or password.

SIREN Code (Système Informatique pour le Répertoire des Entreprises)

This stands for the Electronic List of Enterprises. The SIREN code is assigned to a company when it registers as a business with the French government, and identifies the purpose of the establishment for regulatory reporting purposes in France.

SIRET (Système Informatique pour le Répertoire des Établissements)

This stands for Electronic List of Entities. In France the SIRET is an identifying number given to a French business by the INSEE, an official statistics and economics organization in France. The SIRET number is a combination of the SIREN and NIC numbers. This number is used by the tax and social security authorities to identify a business enterprise and its entities.

Site Tree

In PeopleSoft eStore, a hierarchical structure that controls navigation, as well as content and behavior within the header, footer, and left margin areas of the web page template.

Slice

The span of time into which an element is segmented as a result of element segmentation. Unlike a segment (or period), a slice does not represent a separate gross-to-net process since it affects only a limited set of elements within a period or segment. Like segments, slices have their own begin and end dates.

Slice Dimension

A model dimension used to restrict user access to the system. For example, a product manager's access to the system can be restricted to only the products he or she is responsible for by defining "Products" as a slice dimension, and assigning this person the members of the "Products" dimension he or she can access.

Slice and Dice

Another term for multidimensional analysis. When your data has three (or more) dimensions, you can think of it as being arranged in a cube (or hypercube), with each side representing a dimension. When you analyze the data, you "slice" off part of the cube or "dice" it to get to an individual cell.

Slotting

In PeopleSoft Workforce Rewards, a process by which the system establishes the target market compensation rates to use for compensating workers in non-benchmark jobs.

Social Security Number

Nine numeric digits assigned to an individual by the Social Security Administration. Also known as a Taxpayer Identification Number (TIN).

Source

The Source table stores valid journal entry and posting sources. These can include job titles (such as CFO), user IDs (such as CLERK123), PeopleSoft General Ledger processes (such as Consolidations), or other applications (such as PeopleSoft Payables).

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.

Sparsity/Density

A multi-dimensional concept of whether data exists at intersections of dimensions. If a cube has many dimensions, but little or no data in some of those dimensions, the cube is considered sparse. Sparse cubes take up unnecessary disk space and reduce calculation performance. The goal is to create dense cubes and only use dimension intersections where data actually exists.

Special Accumulator

A device that accumulates earnings from different sources for a specific purpose. 401(k), pension and retirement plans use special accumulators. A 401(k) plan might use a special accumulator to calculate a deduction using regular, vacation, and overtime earnings. Special accumulators can add to or subtract from a pool of earnings.

Special Payments

A payment that occurs once or under special circumstances (e.g., back pay interest, lump sum leave, bond refund, longevity bonus, compensatory time reimbursement, death payment, severance pay, separation bonus, etc.).

Special Rates

Higher salary rates for specific grade levels and occupational groups determined by OPM for employees working in specific geographic areas. Each area is assigned a separate Schedule Number.

Specialist

A member of a stock exchange who maintains a fair and orderly market in one or more securities. A specialist or specialist unit performs two main functions. executing limit orders on behalf of other exchange members for a portion of the floor broker's commission, and buying or selling for the specialist's own account to counteract temporary imbalances in supply and demand, preventing wide swings in stock prices.

Specialized Experience

Experience that has equipped the applicant with the particular knowledge, skills, and abilities to perform successfully the duties of the position and is typically in or related to the work of the position to be filled.

SpeedChart

A user-defined shorthand key designating 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.

Spending Limits

See Planning Targets.

Split and Join

In PeopleSoft Demand Planning,the process of subdividing a forecast so that multiple users can make changes to their portions of the forecast. After changes are complete, the portions are joined back into a single forecast.

Split Deduction

Deduction that you create by splitting an existing deduction into two deduction items. The new deduction retains the original item ID with an added suffix number.

Split Shift

Periods of productive time split up by period of non-working time; example. a time reporter comes to work as a busboy for the lunch shift from 12-2 p.m. and then returns to work from 6-8 for the dinner shift (see Scheduling*)*

Split Stream Processing

The matching of a payment's cash information with the payment advice information when they have been received as separate transmissions through EDI and lockbox.

The uniting of the payment cash with the payment advice when they have been received as separate information through EDI.

Spokesmen Committee (Sprecherausschusse)

In Germany the Spokesmen Committee represents the interest of the management in your company before the ownership. The Spokesmen's Committee is consultative in nature, although they play a co-determination role on individual employment contracts, hiring, and dismissals. They also play a role in monitoring employment fairness, equity, and non-discrimination in terms of nationality, race, religion, sex, and age.

Spouse Demonstration J&S

In the PeopleSoft Pension Administration system, an informational-only form of pension payment that tells what the spouse's total benefit would have been if the retiree had chosen the spouse as the beneficiary rather than a nonspouse beneficiary. You cannot pay pension benefits based on this form because it is informational only.

Spouse Eligibility Alias

In PeopleSoft Pension Administration, a Custom Statement that defines any criteria that must be met before the plan will provide an Automatic Spouse Benefit. For example, the plan may require that the employee and spouse be married a full year before they are eligible for an automatic spouse benefit.

Spouse Eligibility Statement

See Spouse Eligibility Alias.

Spread

Depending on the context, either (1) the difference between the bid and asked prices for an over-the-counter stock, or (2) the difference between an option's exercise price and the market price at the time of exercise (i.e., the profit component of the exercise).

SQL Objects

Used to create rules that are more complicated than templates or actions and conditions allow—select statements, insert statements, table joins, and sub-queries

ST (Strategic Trust)

See Strategic Thrust

Staged Date

The date an item was received into the inventory business unit.

Staging ID

An identifier for a putaway plan. The inventory system sequentially assigns Staging IDs when it creates the putaway plan.

Standard Cost

A predetermined, fixed cost associated with an **Inventory Item** or **Forecast Item**, representing detailed estimates of each element of cost entering into the purchasing or manufacturing of an item. Standard cost is used when minor variations in an item's cost are not needed. The use of standard costs also enables management to determine how much an item should cost (Standard), look at how much it does cost (Actual), analyze the differences between the two and their causes (Variances), and compute economic order quantity.

Standard Form (SF)

A standardized form for interagency use by the Federal government. The SF prefix is the most common but not exclusive one in usage.

Standard Price

In PeopleSoft Demand Planning, the standard selling price associated with a **Forecast Item**. The price can be introduced into the system directly in forecast item maintenance or indirectly using the demand transfer interface. At higher levels in the view where there is no standard price available, the summarization function can be set up to develop one.

Standard Unit of Measure

The smallest unit of an item that a PeopleSoft application tracks.

Startup Data

In PeopleSoft Pension Administration, accrued Service, Cash Balance Account, or Employee Account data loaded into the system in the form of an opening balance and "as of" date. The alternative would be to load the entire accrual history.

State Record

The State Record is a PeopleSoft record, keyed by process instance, that must be created and maintained for each Application Engine program. The State Record defines the fields that an Application Engine program uses to pass values from one SQL statement to another.

Static Group

An employee group in Time and Labor that enables you to control its creation and maintenance. The group remains the same at all times until you change it.

Static Policy Controls

Determines how a static (versus time-phased) **Inventory Policy** is to be calculated. Static controls use period and average methods and their arguments.

Statistical Account

An account that has an associated unit of measure, used for tracking and monitoring statistical data. For example, the Workstations account uses EA (each) as a generic unit of measure, while the Floor Space statistical account might use square feet and the Work Days account would use days.

Statistical Code

The unit of measure used for tracking and monitoring statistical data. For example, using a statistical code of WS may represent the number of Workstations.

Statistical Forecast

In PeopleSoft Demand Planning,, a forecast developed at each level of the forecast pyramid and that considers the item's history in isolation.

Status Checking

In PeopleSoft Projects, a control feature that can be applied to transactions coming into Projects from cost feeder systems. If the incoming transaction does not conform to predetermined status and analysis conditions, an online warning will display or the transaction will be rejected.

Status Position Code

A code that identifies the various conditions of a position, e.g., frozen, classified, etc.

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

A secondary level or subcategory within the primary pay level (depending upon pay plan, different employees may have a different number of steps within their primary pay level).

Step Progression

In PeopleSoft Workforce Analytics, a pay increase granted to an employee or group whose salary plan includes steps within grades. Each step increase is a step up the pay range for the employee.

Stock

In corporate finance, the form in which an owner's interest is represented, distributed in units known as shares.

Stock Administrator

An individual who administers and manages the corporation's benefits and/or equity compensation plans. This individual serves as the contact for transfer agent and broker inquiries. Stock Administrators manage Stock Option Plans, Employee Stock Purchase Plans, Restricted Stock Award Plans, and Stock Bonus Plans.

Stock Appreciation Rights (SAR)

A contractual right to receive, either in cash or employer stock, the appreciation in the value of the employer's stock over a certain period of time. A SAR can be used alone or in tandem with Incentive Stock Options (ISO/SAR) or Nonqualified Stock Options (NQ/SAR). PeopleSoft Stock Administration supports only tandem SAR's.

Stock Awards

Stock allocations that are processed in the Manage Variable Compensation business process. Stock Administration creates stock grants from finalized stock awards.

Stock Exchange

An organized marketplace in which bonds, stocks, and common stock equivalents are traded by members of the exchange, acting as agents (brokers) and as principals (dealers or traders). Such exchanges have a physical location where brokers and dealers meet to execute orders to buy and sell securities. Each exchange sets its own requirements for membership.

Stock Option

A contractual right granted by the company, generally under a stock option plan, to purchase a specified number of shares of the company's stock at a specified price (the exercise price) for a specified period of time (generally five or ten years). Assuming that the exercise price is the

same as the fair market value on the grant date, the option will become more valuable if the fair market value goes up, because the option effectively gives the optionee the right to buy stock in the future at a discount.

Stock Price

The price per share of a company's stock. See, also, "share price.

Stock Purchase Participant

An individual who participates in the corporation's Stock Purchase Plan.

Stock Purchase Plan

A type of broad-based stock plan that permits participants to use payroll deductions accumulated over a period of time to acquire stock from the company.

Stock Split

A change in the capitalization of an issuer that increases or decreases the number of securities outstanding, and adjusts the value of the securities accordingly, without a corresponding change in the assets or capital of the issuer. For example, if an employee has options to purchase 25 shares at $10 per share and the company has a 2-for-1 stock split, the employee thereafter has the option to purchase 50 shares at $5 per share.

Stock Swaps

A payment method that can be used to cover the cost of the exercise price and taxes depending on whether it is allowed by the plan. When an employee elects to exercise a stock option by means of a stock swap, they surrender already-owned shares of stock to pay the total required option exercise price and/or taxes for the option being purchased. The surrendered shares are usually valued at the fair market value of the company's stock on the date of exercise.

Stock Trading Symbol

The three or four letter symbol used to identify a company's stock on the stock exchange where it trades. Also known as a "ticker symbol".

Stock Withholding

A cashless method of satisfying the withholding taxes due upon the exercise of a stock option by authorizing the company to withhold from the shares being exercised a number of shares equal to the taxes.

Stockholder of Record

Person or entity, often a broker or the Depository Trust Company, named on the issuer's or transfer agent's stock record books as the owner of shares held in "street name." The stockholder of record acts in part as a way of safekeeping stock certificates that might otherwise be lost by the beneficial owner, and also in order to keep the identity of the beneficial owner confidential from the company.

Stock-In Probability

A replenishment option for defining transfer parameters for PeopleSoft Demand Planning or Inventory Planning upload files. The option is the percentage of time you want to have the item on hand for the **Business Unit** and is used to calculate safety stock.

Stop Time

Out punch

Storage Area

A division of a **Business Unit** used to store material and to track **Inventory Transaction**. Storage areas might include shipping and receiving docks, staging areas, warehouse zones, and inspection and quality control departments. Each storage area can be divided into a maximum of four levels, with each level representing a physical subdivision of the area.

Storage Level

A hierarchical subdivision of a storage area.

Storage Location

The combination of a storage area and that area's most detailed storage level. This is the smallest definable physical space within an **Inventory Business Unit**.

Strategic Initiatives

In PeopleSoft Balanced Scorecard, actions the organization must take to implement strategy. May be temporary or short-term in nature.

Strategic Thrust (ST)

In PeopleSoft Balanced Scorecard, four to five statements or paragraphs that summarize the core components of an organization’s strategy. Strategic thrusts describe the key areas across which a scorecard is balanced. They are themes or goals your organization is striving to achieve; more specific descriptions of what you must do to achieve that goal are defined by critical success factors. Key performance indicators may be attached to strategic thrusts as long as there aren’t critical success factors below them, but typically strategic thrusts aren’t directly associated with key performance indicators.

Strategy Tree

In PeopleSoft Balanced Scorecard, the hierarchical relationships of the objectives your organization is striving to achieve. Used as the foundation for a scorecard, and typically balanced across four major categories: Financial, Customer, Learning and Growth, and Internal Processes. These are made up of Vision, Strategic Thrusts, and Critical Success Factors.

Stratification Engine

A support module that structures the volume of financial accounts and balances at a large financial institution to a manageable scale for processing by the PeopleSoft Funds Transfer Pricing (FTP) and PeopleSoft Risk Weighted Capital (RWC) applications. It categorizes data by a range of values and summarizes data based on rules you define for FTP and RWC.

Stratification Wizard

Stratification Wizard is a tool you can use to quickly create new stratification rules or update the existing rules. Stratification Wizard prompts you for each of the possible source and destination fields, grouping operations, and summarization actions to be performed. Stratification Wizard enables you to stratify your data according to tiers, discrete values, periodic increments, and numeric increments. It also enables you to leave the data aggregated.

Streams

An optional feature that enables you to reduce processing time by processing groups of payees simultaneously.

Street Name

See "Stockholder of Record".

Stretch

In a Goals Matrix performance scale (In PeopleSoft Workforce Analytics), this is the level of performance for which an employee achieves maximum pay out. Performance above this level receives no greater pay out.

Strike Price

The price per share which must be paid in order to exercise the stock option. The strike price is typically the fair market value of the stock on the grant date. Also known as the "exercise" or "grant" price.

String constant

String constants are delimited in PeopleCode by using either single (‘) or double (") quote marks.

Strip Funding

One of several methodologies used by PeopleSoft Funds Transfer Pricing (FTP) to derive maturity when calculating FTP rates based on matched maturity funding. This approach matches the projected cash for the instrument in each time period, with a specific cost of funds rate for that cash flow. The FTP rate for the instrument is then calculated by weighting the cost of funds rate for the cash flow in each time period by the term of the cash flow.

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.

Style File (Verity)

Collection style refers to a set of configuration options that are used to create the indexes associated with a collection. A collection has one collection style and it is defined in a set of style files before creating the collection.

SubCustomer Qualifier

A value that groups customers into a division for which you can generate detailed history, aging, events, and profiles.

Sub-Process Section

A type of section you can add to a process list. Sub-process sections are especially useful for performing iterative processes such as gross ups (calculating the gross amount for a given net amount). You can include conditional logic within a sub-process section.

Subscription

The process of mapping fields, selecting data parameters and submitting the information to an outside vendor.

Substitute Item

In PeopleSoft Manufacturing, an item that can be used when there are no primary components available in inventory or when there is a long-term shortage of the original item. The substitute item can be defined at three levels: setID, business unit/item, and bill of material/engineering bill of material.

Subtask

A lower-level Planning task in a schedule's hierarchy that rolls up into a parent task. For example, an operation performed on a production ID would be a subtask of the production order.

Summarization Process

See Rollup.

Summary ChartField

A feature for creating 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 Forecast

In PeopleSoft Demand Planning, a type of forecast that results from adding up the adjusted forecast totals from the next lower level, meaning the sum of the children's forecasts for the parent. The summary forecast at level one (1) is always zero since there is not a logical lower level.

Summary Ledger

An accounting feature used primarily in allocations, inquiries, and PS/nVision reporting to store combined account balances from detail ledgers. They increase the 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. In effect, 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.

Summed Adjustment Type

When the system finds multiple summed discounts, they are added together, and applied once.

Super Tree Utility

A PeopleSoft Enterprise Warehouse utility that enables you to combine multiple effective dates of a tree into one. This super tree contains all tree changes for a certain period of time enabling you to analyze effective dated trees more easily.

Supplemental IRR

An IRR type used when a federal employee's retirement deductions were originally under-reported. An agency can create a Supplemental IRR to report the difference in the retirement deduction amount to the Office of Personnel Management (OPM).

Supplemental Tax Method

A payroll tax calculation method that uses a straight percent rather than allowances. The percentage depends on state requirements.

Supply Chain Warehouse

See Warehouses.

Support Costs

Activity costs not directly connected to production.

Support Modules

The support modules are a collection of engines and analysis models that derive values, rates, financial calculations, and prices. PeopleSoft Analytic Applications take this data and use it for further processing. The support modules perform processes that are used in the financial services industry. They are only utilized by two of the PeopleSoft Analytic Applications: Risk Weighted Capital (RWC) and Funds Transfer Pricing (FTP).

Support Team

A group of people working together to sell to and/or support a customer. You can assemble support teams and associate them with ship to customers, quotes, sales orders, and **Buying Agreement**.

Supporting Element Overrides

Provide a mechanism to override various supporting element types, such as brackets, dates, durations, formulas, and variables, at various different levels.

Supporting Elements

Supporting elements are building blocks for other elements. In PeopleSoft Global Payroll, they are used in combination with other elements to create rules. They are not stand-alone. Typical supporting elements are arrays, brackets, rounding rules, and fictitious calculations.

Suspend Exercise

As a condition of a leave of absence, a company may stipulate to restrict the exercise of shares during the leave or for a period of time. Only applicable if the Stock Action is LOA.

Suspend Vesting

As a condition of a leave of absence, a company may stipulate to suspend vesting of shares. Only applicable if the Stock Action is LOA. If a company does not Suspend Vesting then the Vest Deferral Grace Period and Service Rule are not applicable.

Suspended Item

In PeopleSoft Demand Planning, an item suspended by the system. The suspension is due to the lack of demand for the number of periods defined for the item's **Control Group**.

System Element

In PeopleSoft Global Payroll, system elements are delivered and maintained by Peoplesoft. There are two types of system elements: database system elements and system-computed elements. Database system elements contain payee-related data that can be used frequently in a calculation, such as department ID, location, and personal data. System-computed elements are automatically populated by the payroll process.

System Functions

A list of all activities that the system supports, along with their associated General Ledger distributions.

System-Defined Count

The PeopleSoft Inventory feature that employs user-defined criteria to begin the count creation process.

System-Defined History

Any statistical information updated by the posting and aging programs, maintained to reflect customer credit standing.

T

T+3

The obligation in the brokerage business to settle securities trades by the third day following the trade date. "Settlement" occurs when the seller receives the sales price (less the broker's commission) and the buyer receives the shares.

Table

The underlying format in which data is stored by columns (fields) and rows (records, or instances).

Table Lookup

A utility in PeopleSoft Pension Administration that finds an unknown value based on a known one in your data set. For example, you can look up an interest rate based on a year, or an actuarial factor based on an employee's age.

TableMaps

In the PeopleSoft Enterprise Warehouse, TableMaps define the physical relationships between related tables. TableMaps allow you to define "families" of related tables and the columns that define the key relationships between the tables.

TableSet

A group of rows across control tables identified by the same SetID.

TableSet Sharing

Specifying the control table data for each business unit so that redundancy is eliminated.

Tardy

The circumstance when a time reporter reports for work after the scheduled start time

Target

In a Goals Matrix performance scale (In PeopleSoft Workforce Analytics), this is the performance level your organization establishes as the norm for performance and pay out.

Target Cost

A desired target cost (for production, engineering or marketing) is found by subtracting the desired profit margin from a competitive or estimated price.

Target Currency

The value of the entry currency or currencies converted to a single currency for budget viewing and inquiry purposes.

Target End Date

The intended end date for employee schedules in Time and Labor reporting. You establish a target end date, and depending on the work schedule templates, the application determines what the actual end date should be.

Target Grade

Highest obtainable grade for a position.

Target Matrix

In PeopleSoft Workforce Rewards, the Target Matrix defines the level of award to be paid based upon a predetermined level of performance that a defined measure is evaluated against.

Target Plan

In PeopleSoft Workforce Rewards, a plan for distributing compensation awards, in which the level of the award is linked directly to a predetermined level of performance that a defined measure is evaluated against. For example, a Target Plan for a Sales group might be as follows: "As a group, increase the business unit's net income by 10% and each member of the group will be eligible for an award equal to 5% of base pay."

Target Rate

In PeopleSoft Workforce Rewards, calculated market rates you choose to use as new target pay rates for the jobs in your organization. Think of these target rates as pay guidelines, in support of your company's overall pay strategy.

Target View

In PeopleSoft Demand Planning, the of the views being reconciled during the **Cross-View Reconciliation** process. During the process, fields for reconciliation are defined for both a source and target view.

Tariff

In Germany a Tariff is a contract between the employee's unions and the employers' association, defining labor agreements on issues such as standard working hours, income, and vacation. This contract is valid for all of Germany for the business or industrial sector the company is working in (such as ***Banking*** or ***Metal***).

Tariff Area

In Germany, additional labor agreement terms beyond those in the Tariff, such as salary plans or employee reviews, can also be applied based upon the Tariff Area. The Tariff Area is often split along regional lines (such as ***Bavaria*** or ***Berlin***).

Task

See **Manufacturing Task** and **Planning Task**

Task

A piece of work assigned to or demanded of a person; a unit of work (see Time Reporting)

Task Entity

Individual component of a task; for example, Project ID, Activity ID, Work order, Department, Company, Business Unit (see Time Reporting).

Task Profile

A way of viewing or establishing where to allocate employee task information for a day and time. The task profile fields that appear on the page are established by the *Task Profile Template.*

Task Profile

Entity that establishes the default values for optional and required task elements. This can be for single or multiple tasks. (i.e. default values based on hours allocation, percentage distribution, equally distributed or by prompt) (see Time Reporting)

Task Rules

A methodology that is applied to scheduled, reported and payable time to allocate or redistribute task assignments (see Time Administration)

Task Transfer

Department transfer

Task Values

The customer defined value for a specific task element i.e. Customer 1, project 1, etc (see Time Reporting)

Taskgroup

Identifies the default time reporting templates, task template, and task profile(s) for time reporters that share the same task reporting requirements.

Taxable Benefits

Any employer contributions that are subject to Federal Withholding Tax.

TDS (Transfert de Données Sociales)

In France the TDS is a social security transfer report, submitted on magnetic media to the government.

Team Member

An individual who is part of a support team. Each team member may be in a commissionable or non-commissionable role.

Technical Scenario

In PeopleSoft Enterprise Warehouse technical scenarios allow you to set up the object type values that the Resolver uses to chunk the record/TableMap you'll resolve. Technical scenarios allow technical users to define chunking criteria that enable chunks to run in parallel. This allows for parallel data processing.

Template

A portal template is simply HTML code, associated with a web page, to define the style and layout of the page. Templates allow a developer to build an HTML page by combining HTML from a number of sources. Templates do two basic things: define the layout of the page, and define where to get HTML for each part of the page.

Template Pagelet

One piece of an overall template. For example, in a given template, there may be one template pagelet for the universal navigation header and one template pagelet for the target content.

Temporal Constraint

A relationship between Planning tasks that defines their sequence and timing in a schedule. Temporal constraints cannot be violated by the Optimizer. PeopleSoft Planning constraints include finish to start, start to start, finish to finish, start to finish.

Temporary Continuation of Coverage (TCC)

The TCC program, as prescribed by the OPM, requires Federal agencies to provide to separating Federal employees the opportunity to temporarily continue their FEHB coverage for up to 18 months (unless involuntarily separated because of gross misconduct), provided the individual pays the full cost of coverage, including both the employee and government share and a two percent administrative charge. Agencies may elect to provide this service in-house or enter into cross-servicing agreements with another Federal agency.

Tenor

Used by the PeopleSoft Funds Transfer Pricing (FTP) application to refer to the maturity of an instrument. It represents the length of time an instrument is available as either a source or use of funds. The FTP application calculates the transfer price for an instrument, based on the marginal cost of funds of similar liquidity and tenor.

Term Certain and Continuous Payment Option

See Certain and Continuous Payment Option.

Term Certain Payment Option

See Certain Only Payment Option.

Termination

A transaction in which an employee ceases to be an employee of the corporation.

Threshold

In a Goals Matrix performance scale (In PeopleSoft Workforce Analytics), this is the minimum threshold for adequate performance, the designated level of performance below which it is inappropriate to pay incentives.

Think-time process

Think-time functions suspend processing either until the user has taken some action (such as clicking a button in a message box), or until an external process has run to completion (for example, a remote process).

Three-Tier

A three-tier architecture introduces an intermediary application server between the client workstation and the database server to improve performance. Within PeopleSoft, the application server sends the SQL to the database and then returns results to the client in the form of lightweight Tuxedo messages.

Threshold Checking

In PeopleSoft Projects, a control feature that can be applied to transactions entered directly into Projects or integrated into Projects via the INTFC_PROJ_RES table. If the transaction exceeds a predefined tolerance, a warning will display or the transaction will be rejected.

Thrift Savings Plan (TSP)

A voluntary retirement savings and investment plan for Federal employees administered by the Federal Thrift Investment Board.

Ticker Symbol

The three or four letter symbol used to identify a company's stock on the stock exchange where it trades. Also known as a "stock trading symbol".

Tiers

In the financial services industry, Tiers are ranges that you set up for stratifying your instrument data (such as amounts, rates, and numbers) into specific groups. You define Tiers within Stratification Engine.

Time Administration

A process which provides four [separate] different online tools for creating, maintaining, and applying an organization's compensation, task, and exception rules to both reported and scheduled time. templates, actions and condition, SQL objects, and user exits

Time and Labor Period

A distinct, configurable period of time used by all the PeopleSoft Time and Labor processes (see Time Reporting)

Time and Labor User

Either a Time Reporter or a Time Manager

Time Capture Device

Third party system or methodology for collecting elapsed or time capture device time, i.e., time capture device, IVR, Fax, etc. (see Time Reporting)

Time Collection

A Time and Labor feature that collects positive and exception time reports, applies appropriate business rules and edits to the reported time to ensure validity and reasonableness, and returns errors and questionable items to the time reporter for correction or scrutiny. Time collection is also responsible for scrutinizing future (previously) posted time information for correctness when those reports are ready for use.

Time Collection Device

A group of time collection device lumped together and named for ease of assignment to employees. In other words. clock group 1 is made up of clock 1, and clock 2. Employee 123 is assigned to clock group 1 and can then punch in at either clock 1 or clock 2. (see Understanding Time Collection Device)

Time Collection Device time

Reporting time by recording actual starts and stop times (see Time Reporting)

Time Dimension

Determines how date-related information is presented in a **Cube View**. This dimension defaults to a two-level hierarchy consisting of the **Inventory Policy** year and a standard period, such as monthly.

Time Fence

In PeopleSoft Planning, a user-defined parameter that specifies the business rules to be used in the generation of the plan. PeopleSoft Planning time-fence types include start of time, end of time, planning close date (demand time fence), purchase order fence, leveling fence, action message cutoff, and planning time fence.

Time Manager

An individual who supervises Time Reporters

Time Period

A period of Time used in Time and Labor rules processing. You can categorize time periods in terms of days, weeks, or months. You establish day, week, or month-type periods for use when you apply rules for compensation, holidays, and so on.

Time Report

A payroll time and/or labor distribution time report for an employee for any date within the employee's current period.

Time Reporter

Any employee or contractor for who time is reported or generated in PeopleSoft Time and Labor.

Time Reporter Information

Values associated with the Time Reporter that are displayed when entering or viewing reported time and facilitate the processes of Time Reporting and Time Management (see Time Reporting)

Time Reporting

Any information required by a business unit that can be attributed to an individual employee (worker/contractor) and can be expressed in hours.

Time Reporting Code

A hybrid of two PeopleSoft objects: the Payroll Earnings Type and the Human Resources Absence Type. The Time Reporting Code represents the level at which a business actually needs to track employee time to support all of its administrative and compensation needs.

Time Reporting Code Type

Categorization of a time reporting code. Valid categories include. units, amounts, hours or a combination of hours and amounts (see Time Reporting)

Time Reporting Group

See Group [Time and Labor].

Time Segment

For Service, Cash Balance Accounts, and Employee Accounts, employees can accrue benefits differently at different times. The period of time during which employees use a particular rule is that rule's time segment.

TimeSpans

Relative periods, 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.

Tolerance

In PeopleSoft Projects, a value that is defined at either the project or activity level as either a percentage of a project's funding or an actual amount.

In commitment control, the percentage over budget that you allow, excluding revenues applied to increase budget limits, before the system creates an exception.

Total Authorized But Unissued

The combined total number of shares from Shares Available to Issue plus Total Options Outstanding.

Total Compensation

In PeopleSoft Workforce Analytics, this is generally the officially recognized compensation provided an employee in the course of their employment with an organization; includes both direct compensation and benefits compensation.

Total Compensation Management

The ability to track and report on all types of cash programs, non-cash programs, benefits and deferred compensation for all current workers, ex-workers and individuals associated with ex-workers who receive compensation due to the employment of the ex-worker.

Total Non-Compensation

In PeopleSoft Workforce Analytics, generally, this is the often unrecognized compensation an employee receives in the course of their employment with an organization; it includes Learning and Development compensation such as training, and Workplace Environment Compensation such as telecommuting privileges or other prerequisites.

Total Options Outstanding

The number of company shares currently held by shareholders as tracked by the transfer agent. Derived by using the number of Prior Outstanding plus Grants, less Exercises and less Cancellations.

Total Rewards

In PeopleSoft Workforce Analytics, this is the total rewards provided to an employee by their employment with an organization; it includes their officially recognized total compensation, and less often recognized total non-compensation.

Tour of Duty

The scheduled days and hours per day of attendance at a duty station for an employee.

Tracking Signals

PeopleSoft Demand Planning,a forecasting tool that detects bias in the forecast and provides an early warning of an unstable forecast. There are six tracking signals associated with each **Forecast Item** that correspond to the six most recent historical periods.

Trade Payment

An authorization for a customer deduction in a Promotion application.

Training Report 2483

The Training Report 2483 is a French regulatory report used to declare vocational training your company has provided to your employees. It is also known as the Declaration 2483 Report. The purpose of the report is to receive tax deductions from the government based upon the amount of money your company has spent on training.

Transaction

A named command with optional named and typed inputs and outputs. The associated external system or the Business Interlink Plug-in understands this command. The types of inputs and outputs are based on a set of generic types.

Transaction

A named command with optional named and typed inputs and outputs. The associated external system or the Business Interlink Plug-in understands this command. The types of inputs and outputs are based on a set of generic types.

See also **Inventory Transaction** or, for PeopleSoft Projects, **Resource Transaction.**

Transaction catalog

Lists transactions used to interface to the external system.

Transaction Code

In PeopleSoft Projects, an additional field on each resource transaction that is used in conjunction with accounting entry templates. Transaction codes enable you to deal with exceptions to your accounting entry templates without having to create additional transaction types. You can set up separate accounting entry templates for resource transactions containing the transaction codes you create. The accounting entry templates for those resource transactions lines can then use the same transaction types, but specify different accounts.

In PeopleSoft Asset Management, transaction codes identify special asset transactions and are used in conjunction with transaction type to create accounting entries.

Transaction Code

Identifies what action has taken place against the position.

Transaction Costing

See Multidimensional Costing

Transaction Currency

In the financial services industry, the original currency in which a company conducts its business activities. When a company has multinational operations, it may use different transaction currencies. These are translated to the base currency for consolidation and reporting of financial results.

Transaction Date

The date a transaction actually occurred as opposed to the date the transaction is recognized—the accounting date (although the two dates can be the same).

Transaction Dated

Data aggregated over a date range.

Transaction group

The package can contain one or more transaction groups. Each transaction group is a set of transactions of the same type, with the same trading partners involved.

Transaction Loader

The SQR in PeopleSoft Asset Management that transfers load lines from the Loader tables into the PeopleSoft Asset Management Tables as assets and open transactions.

Transaction Tables

In the PeopleSoft Enterprise Warehouse, these are tables that contain dynamic information and are keyed by business units.

Transaction Type

The building blocks of accounting entry templates in PeopleSoft Asset Management and Projects. For each transaction type you create you define specific transaction lines. The transaction lines are then transferred into accounting entry templates. In the accounting entry templates each transaction line is assigned a specific general ledger account.

Transactional System

A business application for performing the business transactions that keep your company running. Transactional applications, and the databases that support them, are optimized for quick transaction processing. Because they are constantly changing and are not optimized for data retrieval, transactional system databases are not usually the best source of data for analysis.

Transfer Agent

An individual or firm who that keeps a record of your shareholders and the number of shares they own. Transfer Agents also issue new share certificates and cancel old certificates. Unlike Brokers, Transfer Agents are not responsible for selling stocks. Instead they are primarily concerned with maintaining records on all stocks which your company has issued.

Transfer Forecast

In PeopleSoft Inventory Planning, a Generation process option that transfers the forecast from the target view in Demand Planning forecasts. The process only transfers items from Demand Planning that have been set to update the **Inventory Policy**.

Transfer Punch

The start of a work period that specifically denotes a change in task and usually compensation-related characteristics

Transfer Type

An interunit transfer setting PeopleSoft Production Planning and Enterprise Planning use to determine where it will obtain item data for transfer tasks. If the type is a supply or demand transfer task, the Planning engine only processes the transfer item for a single location, reducing the time for plan processing. If the transfer type value is both, the Planning engine processes the transfer item using data from both the To and From units.

Transfer Worksheet

A work space for transferring an open item from one customer to another.

Transferable Stock Options

Options that may be transferred by the optionee, generally only to a family member or to a trust, limited partnership or other entity for the benefit of family members, or to a charity.

Translate Table

A system edit table that stores codes and translate values for the miscellaneous fields on the database that do not warrant individual edit tables of their own.

Translate Table

A system edit table that stores codes and translate values for the miscellaneous fields on the database that do not warrant individual edit tables of their own. In most cases PeopleSoft maintains the Translate Table.

Transport Rate

The Transport is a statutory deduction in France. Each establishment has a rate, and the URSSAF notifies establishments of this rate on a yearly basis. This deduction is used by the region to subsidize transportation, and maintain and build roads.

Transportation Lead Times

The transportation lead time is the in-transit interval from the date and time a shipment leaves your warehouse (**Inventory Business Unit**) to the date and time it arrives at your customer's receiving dock. The transportation lead time is used in calculating the scheduled shipment and scheduled arrival dates on the order when you enter either a requested arrival date or a requested shipment date.

Travel And Relocation Date

Length of time an employee must remain in the Government after the Government has paid to relocate him/her from one official duty station to another or for initial appointment.

TRC Program

A program that runs the level at which an organization actually needs to track employee time to support all of its administrative and compensation needs. TRCs are assigned to TRC Programs, which are ultimately assigned to workgroups. Multiple Workgroups can share these TRC Programs.

Treasury Interface files

These are DOS-based files generated by PeopleSoft in accordance with FMS file layouts for transmission of payment data to one of the FMS' Regional Financial Centers.

Treasury Position Code

In the financial services industry, this is a lookup code used for off-balance sheet treasury position accounts, such as foreign exchange, derivatives, precious metals, or any other account position that is the result of trading room and treasury operations.

Treasury Stock

Shares of a company's stock that have been repurchased or otherwise reacquired by the company and are "held in treasury." Whether the treasury shares count as "issued" or as "outstanding" shares of the company is a matter of state corporate law. Generally, a company may not vote its own shares held in treasury.

Treasury Stock Method

The method of calculating primary and fully diluted earnings per share when common stock equivalents such as unexercised stock options exist. Required under generally accepted accounting principles.

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.

Tree Compare Utility

A PeopleSoft Enterprise Warehouse utility that enables you to compare effective dates for trees. The results page shows nodes that have been added, deleted, or moved from one parent to another. You may also view the detail objects that have changed.

Tree control

Tree Control is a hierarchical search tool that you can imbed in a panel. Tree Controls give the user a view of hierarchical data structures and enable them to drill down through the hierarchy to a particular row of data.

Tree Denormalizer

The Tree Denormalizer Application Engine process converts trees into multi-column data format so they can be used by third-party OLAP or ROLAP tools.

Trigger

See Event Trigger.

Trustee Extract

A PeopleSoft Pension Administration data extract containing data that a third party needs in order to produce pension checks.

Turnover Costing

In PeopleSoft Workforce Analytics, this is a calculation of the cost to the organization of employee turnover, in dollars.

Turnover Rate

In PeopleSoft Workforce Analytics, the rate that employee's are leaving the company.

TUXEDO

BEA's middleware product used to manage transaction queues, server process initiation, system administration, time-outs, data encryption, compression, logging and other application server processing.

Two-Tier

A two-tier architecture refers to the traditional client/server model in which a client workstation connects to and sends SQL directly to the database server.

Type of Appointment

Indicates the specific type of appointment, e.g., part-time permanent, full time temporary, etc.

U

Underlying Security

The security underlying a stock option that an optionee has the right to buy, or the security underlying a convertible security.

Underpayment Adjustment Limit

The maximum amount or percent above which underpayment adjustments are not allowed for a given business unit.

Underwater Option

When the current market price is below the option exercise price. When an option is underwater, it would cost more than the underlying stock is worth to exercise the option. Such options are also described as being "out-of-the-money."

Underwriter

An investment banking firm that actually buys the shares from the company in a public offering and then resells them (at a slightly higher price) to its customers.

Unexpected Losses

In the financial services industry, these occur when the economic capital is exhausted and the insolvency rate is exceeded. Unexpected losses are determined by a targeted insolvency rate (confidence level); for example, a 99.7% confidence level indicates that there is a 0.03% estimated probability that the unexpected losses will exceed economic capital (or shareholder equity).

Union Code

Part of a group of defaults assigned to job codes. Union code may be used by human resources to group similar jobs or bargaining units together, dependent on individual company parameters.

Unit Code

In the financial services industry, Unit Code is used as an alternate means of measuring the relative size of companies participating in external surveys. A typical measure would be the number of employees in a company. The concept of unit is generic enough that the units can be other measures besides number of employees. For example, in the hospital industry the unit could be the number of hospital beds. Or in the hotel industry the unit could be the number of rooms.

Unit of Measure (UOM)

A type of unit used for quantifying in PeopleSoft systems. Depending on the application, units of measure might describe dimensions, weights, volumes, or amounts of locations, containers, or business activities. Examples include inches, pounds, workhours, and standard cost dollars.

Unit of work

Each transaction group includes one or more individual units of work. A unit of work is a single transaction that you want to commit or rollback as a whole.

Unitize Assets

The process of unitizing a single load line, usually originating from a different application, into multiple assets in PeopleSoft Asset Management.

Univariate Forecasting Technique

In Enterprise Planning and Simulation, the Univariate Forecasting Technique is a forecasting method that uses only the recorded history for the value to forecast its future.

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.

Unscheduled Punch

A punch that is made be a time reporter who was not scheduled (see Time Reporting)

Unvested Shares

Unvested stock options are options that have not vested and, therefore, are not exercisable.

URI

A URI does not include the query string (the text following a ? on the URL). You can think of it as a subset of the URL that points to the resource, but does not include any parameters being passed to that resource. From the above example, the URI portion of the URL is as follows:

http://serverx/InternetClient/InternetClientServlet

URL

In this document, 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

URSSAF Code

The URSSAF is the body responsible for ensuring payment of Social Security contributions by all French employers.

Useful Life

The amount of time an asset may be depreciated.

User Data

PeopleSoft Demand Planning,data held in user-defined fields. These fields provide for storage of additional data that is not supplied by the standard set of fields in the system. The fields can also become part of the key for the **Forecast Item** at each level within the **Forecast View**.

User-Defined History

A summary of customer receivables activity that is defined by the user.

User-Field Code

PeopleSoft Demand Planning,a definition of a set of user-defined fields that contain data specific to the installation.

V

Valuation

The way a company represents the value of a non-monetary award such as stock.

Value allocation

A process in PeopleSoft Workforce Planning, by which you assign an overall monetary value to a competency strategy for your organization, and assign a weight or importance to the roles,

competencies and accomplishments in the strategy. The system then allocates a breakdown of the value to roles, competencies, and accomplishments in the strategy based on their relative weighting or importance.

Value Object

In the PeopleSoft Enterprise Warehouse, Value Objects are a metadata layer that provides descriptive information about fields and values. Value Objects are used as constants in Data Manager target object definition.

Variable

Temporary storage for use or defined information used in the creation and application of rules (see Time Administration)

Variable [Global Payroll]

An element type that defines and stores values such as a character, date, or number. You can use variables to create generic formulas for situations where you use the same values over and over again in a calculation.

Variable Compensation

In PeopleSoft Workforce Analytics, direct compensation that is not fixed, that is paid out in variable amounts, such as bonuses and commissions.

Variable Plan.

A plan in which either the number of shares and/or the price at which they will be issued is not known on the grant date.

VAT Account Type

A code that identifies the different types of accounting entries that must be created for VAT transactions. These codes are also used to categorize transactions in the VAT transaction table. The account type is used in conjunction with the VAT code and VAT transaction type to determine the VAT ChartFields used for a given VAT accounting entry.

VAT Apportionment

For mixed activity, VAT apportionment is the mechanism that allows you to specify the ratio of taxable activity to exempt activity for individual ChartFields.

VAT Calculation Method

Options are Net or Gross. When calculating VAT at net, the early payment discount is applied to the goods amount before calculating the VAT. The amount of VAT calculated using this method is the amount that is to be paid, regardless of whether the early payment discount is actually taken at time of payment. When calculating VAT at gross, the VAT is initially calculated based on the gross transaction amount. The early payment discount is not taken

into account at this point. However, in some countries an adjustment is made to the VAT amount at the time of payment, if the early payment discount is taken.

VAT Calculation Type

Options include Exclusive or Inclusive. If exclusive, the VAT amount is stated separately from the goods amount. If inclusive, the VAT is not stated separately but is included with the goods amount.

VAT Code

The tax code used to define a percentage the system uses to determine the VAT amount. The VAT code is similar to the sales and use tax code, with a few exceptions. The tax authority tied to the VAT code generally consists of a single authority, and the ChartFields for a VAT code don't reside with the tax authority but are determined by the combination of the VAT code, VAT account type, and VAT transaction type.

VAT Declaration Point

When VAT transaction information is declarable for reporting purposes. Options include Invoice or Payment. If you choose invoice, the system will recognize VAT at invoice time; if you choose payment, the system will recognize VAT at the time of payment.

VAT Entity

The level or entity within an organization at which VAT reporting is performed. VAT entities can be registered for VAT in multiple countries, but only one country can be designated as the VAT entity's home country. VAT and Intrastat reporting information and VAT default information are defined for each country in which the VAT entity is registered. You may also specify any VAT exceptions—either exoneration or suspension from paying VAT—for any country in which the entity is registered.

VAT Exempt Supply or Purchase

A transaction where the product or item is non-taxable or exempt from VAT. No VAT code is associated with the transaction. Although no tax is applied to the transaction, the transaction is still logged in the VAT transaction table.

VAT Exonerated

A transaction where the purchaser has been determined as not subject to VAT. For these cases, there may be an exoneration certificate number tied to the purchasing entity (either the customer or the VAT entity) as proof of exoneration. A zero-rated VAT code should be associated with transactions where exoneration applies. The transaction is still logged in the VAT transaction table, but no tax is applied.

VAT Rebate Percent

Within Canada, for Public Service Bodies, the percentage of VAT that is not normally recoverable but which may be refunded in the form of a tax rebate.

VAT Recoverability Percent

The percent of VAT that's recoverable.

VAT Registration Countries

Country codes associated with a VAT registration number for a particular customer or VAT entity.

VAT Transaction Table

Stores detailed transaction information for VAT reporting. It is the primary source of information for all VAT reports. Each application is responsible for writing to this table and also to a cross-reference table used to link entries in the VAT transaction table with entries within each application.

VAT Transaction Type

Used to categorize VAT transactions according to particular VAT accounting and reporting requirements. The VAT code and the VAT transaction type are used in conjunction with the VAT account type to obtain the ChartFields for accounting entries.

VAT Treatment

A description of how the transaction must be treated for VAT purposes. This is used to determine how VAT defaults are applied, what accounting entries are required, and how and if the transaction is reported on the VAT return.

VAT Use ID

A code used to identify the type of activity in which a purchased good or service will be used, and therefore to determine a recoverability percent and a rebate percent (when applicable) that will be applied to a transaction line. Activities are categorized as taxable, exempt, or mixed. Where activity is mixed, you may associate either the ratio of taxable activity to exempt activity directly with the Use ID, or you may indicate that this ratio is determined at the ChartField level.

VdkVgwKey

A key within a Verity BIF file for every document to be indexed. VdkVgwKey values must be unique across all collections that will be searched in any one application.

Vendor Draft

A draft issued by a vendor. PeopleSoft Receivables generates vendor drafts, provides a flexible worksheet environment for approval management, and enables discounted or standard submission for bank processing. PeopleSoft Payables receives vendor drafts and associates the appropriate vouchers.

Verity

The third-party search engine integrated with the PeopleSoft Portal.

Verity Fields

Verity fields are stored in the collection for retrieval and searching, and can be returned on a results list. Fields are defined in the BIF file and stored in the collection for retrieval and searching, and can be returned on a results list. Fields, like date and numeric fields can be used with the comparison operators (<,<=,>,>=).

Verity Thesaurus

The custom thesaurus consists of lists of synonyms defined in a synonym control file and can be used for synonym searching. After defining synonym lists in the control file, you use the mksyd utility to create a custom thesaurus (a control file which has the .syd extension) that the search engine uses.

Verity Topics

Verity applications can provide end users with predefined search criteria called *topics*. A topic is a named object that represents a concept, or subject area and can be used for synonym searching. It consists of words and phrases grouped together using the Verity query language in a tree-like structure. When provided, topics can be shared by all users.

Verity Zones

Zones are specific regions of a document to which searches can be limited. When the zone filter is used, the Verity engine builds zone information into the collection's full-word index. The index, enhanced with zone information, permits quick and efficient searches over zones. Searching a zone is faster than field searching. Zones are defined in the DAT file. The contents of a zone cannot be returned in the results list of an application.

Version

There can be up to five budget versions for each Budget Center level in a Budgeting Model. Budget versions are used to perform what-if analysis and comparisons of budget amounts before the user selects one version to submit as the Budget Center's budget plan. PeopleSoft Budgeting-specific.

Vest Deferral Grace Period

The specified period of time within which an optionee must return from leave to avoid having the vesting differed. Only applicable if the Stock Action is LOA. Suspend Vesting must be selected for this rule to be applicable.

Vest Deferral Grace Period Service Rule

If the company provides a vest deferral grace period, they may stipulate that only certain individuals are eligible for the grace period based on service with the company. Only applicable if the Stock Action is LOA. Suspend Vesting must be selected for this rule to be applicable.

Vest Immediately

A stock option plan may provide that upon specific types of terminations, or upon a case by case scenario, all unvested shares held by an individual can be made immediately vested as of a specific date. Some companies' plans provide that under certain circumstances, such as retirement, the vesting of option shares accelerates upon termination of employment. When this occurs, you must modify the vesting schedule before you terminate the individual.

Vested Shares

Option shares that are free of any ownership restriction. Generally, vested exercised shares are fully owned by the optionee, free from restrictions and freely tradable.

Vested Termination

The termination of an employee who has a vested benefit. The benefit is deferred until the participant reaches retirement age. The employee is considered “Terminated Vested,” “Term Vested,” or simply “TV.”

Vesting

The method by which a granted option becomes free of all restrictions and the Optionee has full rights to the shares.

Vesting Schedule (Template)

A convenient way to set up the framework for a vesting schedule that can be uniformly applied to individual options. When you grant stock options, you define a vesting schedule to determine the default-vesting schedule for the option.

Vesting Service

The service used to determine an employee’s vesting percentage. Rules for accruing vesting service may be different from rules for accruing other plan service credits.

VETS–100 Federal Contractor Report

This report is required of employers in the United States. It lists federal job classifications, and the number of employees and new hires in the last 12 months who are special disabled military veterans or Vietnam era military veterans. It also provides totals for each job classification of both veterans and non-veterans who hold these jobs.

View

PeopleSoft Demand Planning,a multilevel forecast structure. Each view is associated with a unique view ID and includes information that defines the view and structure type. The three types of views are working, disbursement, and dynamic.

For PeopleSoft Budgeting see Budget View.

Virtual Tasks

In Time and Labor, Virtual task data is associated with a taskgroup profile that defines common characteristics for a given Taskgroup and Task Profile ID. A single row of data is linked to multiple Earnings records for multiple employees. By minimizing the physical storage of daily task data we provide enhanced performance without limiting its functionality.

Vision

In PeopleSoft Balanced Scorecard, the overall mission of an organization. Usually the highest level on a strategy tree. Vision is optional; you aren't required to have a vision component on each strategy tree.

See also Strategy Tree

Volume

Total share volume traded in a stock during market hours.

W

WA (Workforce Analytics)

See PeopleSoft Workforce Analytics

Waiver Of An OPM Qualification Standard

Involves setting aside requirements in a published standard to place an employee in a particular position, usually to avoid some kind of hardship to the employee, such as in cases of RIF or administrative error on part of the agency. Extra training and/or skills development may be needed to help the employee adjust to the new position. Waivers are granted by OPM or an agency, as appropriate, on a case-by-case basis, and do not directly affect other positions in the organization.

Warehouses

A warehouse reporting and analysis solution that supports the specific PeopleSoft business application that warehouse is using. It consists of predefined ETL maps, data warehouse tools, and Data Mart definitions. The warehouses are we deliver are: PeopleSoft Financials Warehouse, PeopleSoft HRMS Warehouse, PeopleSoft CRM Warehouse, and PeopleSoft Supply Chain Warehouse.

Warning Exception

A transaction that exceeds the available funds but is allowed to continue to be posted against the budget. Warnings are informational only.

Warrant

A type of security, usually issued together with a bond or preferred stock, that entitles the holder to buy a proportionate amount of common stock at a specified price, usually higher than the market price at the time of issuance, for a period of years or to perpetuity. A warrant is usually issued as a sweetener, to enhance the marketability of the accompanying fixed income securities. Warrants are freely transferable and are traded on the major exchanges.

WCB

In Canadian provinces the Worker's Compensation Board (WCB) operates as an independent board, and thus would have different requirements in each province. For example, in British Columbia the organization is called the Worker's Compensation Board of British Columbia and in the Province of Quebec, the board is known as Commission de la Santé et de la Sécurité du Travail (CSST).

Weight

In PeopleSoft Planning, a user-defined value for the constraints that can be violated, determining how the schedules score will be calculated. Violations that are more critical to your schedule merit a higher weight.

Weight and Volume Pricing

You can price shipments by weight or volume to create price prices. Weight and Volume pricing requires using estimated shipments.

Weighted Average Cost of Funds

The projected principle payments for an instrument are used to derive a series of matched maturity funding rates, which in turn are used to calculate the overall base PeopleSoft Funds Transfer Pricing (FTP) rate. The Weighted Average Cost of Funds (WACF) method calculates a weighted average FTP rate where each of the funding rates is weighted by the principle payment amount and the term to maturity of the payment.

WFA (Workforce Analytics)

See PeopleSoft Workforce Analytics

WGI Due Date

Identifies the date of an employee's next within grade increase. Current policy is that the step increase is implemented on this date automatically unless prevented by the processing of an unsatisfactory performance appraisal.

WGI Non-Creditable Days

Total number of days that cause the WGI due date to be adjusted forward.

Whole Calendar Month

An instruction telling the system to use every day in each month for this time period. The system fills in the last day of the period according to the information you have entered.

Wildcard

You can replace the right-hand characters in a search field with a percent (%) wild card to query a range of values beginning with the remaining, left-hand characters. For example, by entering '2%' in a six-character field, you will receive a range of available values, such as 200000 through 299999 or 2aaaaa through 2zzzzz.

Window Period

The ten-day period, from the third to twelfth day after public release of a company's financial statement, when insiders may exercise their stock-appreciation rights without violating Securities and Exchange Commission rules for short-term trading.

Windows Client

Traditional PeopleSoft 32-bit client. Windows clients connect to the application server domain (Tuxedo) using a port number (or connection string) specified in PeopleSoft Configuration Manager.

WIP Replenishment Method

Designates how the PeopleSoft Flow Production request is communicated. For a replenishment method of Inventory, the Workflow, Pull Ticket, and Pull List replenishment methods are available. With feeder line replenishment, you can only use Pull Tickets.

WIP Replenishment Mode

Determines how PeopleSoft Flow Production is triggered to generate a replenishment request for an item. Replenishment options include Backflush, Manual, and Kanban Card.

WIP Replenishment Source

Determines where you send your PeopleSoft Flow Production replenishment request and what source supplies your WIP location. Options include Feeder, Inventory, and Vendor.

Withdrawal

An election not to continue participation in a stock purchase plan.

Withdrawal of Contributions

In a pension plan, the act of returning pension contributions, with interest, to an employee who is terminating. An employee who withdraws contributions typically forfeits all service associated with those contributions. If the employee is later rehired, repayment of contributions and interest typically reinstates the forfeited service.

Withholding

A deduction taken by employers out of taxable income of an individual. Typical withholding taxes include federal income taxes, federal social security, Medicare taxes, and state and local income taxes.

Within Grade Increase (WGI)

A longevity-based increase in salary based on predetermined time in grade requirements and acceptable performance.

Work Council (Comité d'Enterprise)

In France it is mandatory for companies with more than 50 employees to elect a Work Council to represent the employees in negotiations with management.

Work Effort

See Activity Type.

Worker

In PeopleSoft Workforce Analytics, workers are defined as anyone who performs functions for the organization, and receives compensation from the organization's operating expense funds in return. Workers can be direct employees or independent contractors. This includes individuals contracting business directly from the company or through an agency.

Work Group

In PeopleSoft Enterprise Performance Management, the work group is a grouping of employees that share a similar activity profile.

Work Period

A Days On/Days Off template; the smallest unit of time that a business uses to communicate with their employees regarding when to be and/or not to be at work (that is, time working and time not working). The work period can be any number of hours. Until clock hour reporting is implemented, the application does not care about the number of hours. The initial Time and Labor product will apply the work period to a calendar day.

Work Queue

In PeopleSoft Demand Planning and Inventory Planning, a feature for reviewing and working with exceptions created during the processing of forecasting and inventory data.

Work Schedule

A template consisting of a sequence of work periods (days) on and off, and the number of scheduled hours per work period. Work Schedules and Work Periods should not be confused with calendar days.

Worksheet

A way of presenting data to the user through a BAM interface that enables users to do in-depth analysis using pivoting tables, charts, notes, and history information.

Work Templates

Work templates describe your employee's work patterns. Work templates could apply to individuals or entire organizations. For instance, 9 AM to 5 PM, Monday through Friday is a fairly standard working week in organizations.

Workday

A 24-hour period rounded by daybreaker with one or more associated shifts (see Scheduling)

Workday Override

A function that allows a Time Manager to override a Time Reporter's schedule for a single workday. For example, Jane's long-term schedule assignment is Monday – Friday, 8.00 to 17.00. Due to an increase in production demand, her manager needs to schedule her to work 7.00 to 18.00 on Thursday, 16 March 2000. Her manager needs to be able to make this change to her schedule in the PeopleSoft Time and Labor system, so when Jane checks her schedule for this week, she'll see the revised schedule.

Worker

Workers can be defined as anyone who performs functions for the organization and receives compensation from the organization's operating expense funds in return. Workers can be direct employees or independent contractors (includes individuals contracting business directly from the company or through an agency).

Workers Compensation

The days an employee is on LWOP due to sustaining an injury or illness while on the job.

Workflow

The background process that creates a list of administrative actions based on your selection criteria and specifies the procedure associated with each action.

Workflow

The background process that creates a list of administrative actions based on your selection criteria and specifies the procedure associated with each action.

Workforce Monthly Report (Déclaration Mensuelle Obligatoire des Mouvements de Main D'oeuvre)

In France, companies that employ 50 or more employees are required to submit the Workforce Monthly Report to the Administrative Division of the Ministry of Work and Social Relations. The report contains workforce information for a given establishment of a company, including

the total number of employees and details of employees who have joined or left the establishment during the month.

Workgroup

A user-defined group of employees who share identical compensation rules. A workgroup may be equivalent to all the employees in a business enterprise, all employees in a Paygroup, all employees belonging to the same Union or Union Local, or all employees who work at a specific work location.

Worklist

The automated "to do" list that Workflow creates. From the Worklist you can directly access the panels you need to perform the next action, and then return to the Worklist for another item.

Worklist

The automated “to do” list which 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.

Works Councils (Betriebsrat)

In Germany, the works councils for your company are internal committees elected by the employees that represent the interests of salaried and hourly paid employees, other than management. Every work location in your company has it's works council (this would be the local works council) and the company as a whole has a central works council.

Work-Study Program

Government or non-government programs supervised work experience related to a student's course of study and are a part of, or a supplement to, education. Federal student-trainee programs are examples of such programs.

X

Y

Yearly Maximum Pensionable Earnings (YMPE)

Amount set by the government upon which Canadian Pension Plan (CPP) contributions are made.

Z

Zero-Based Budgeting

A budgeting option that builds a budget from the ground up starting with zero values. This is in contrast to an incremental budget that is based upon using prior year actual or budget values as starting point. PeopleSoft Budgeting-specific.

Zero-Rated VAT

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.

Zip Code

The term for postal codes in the United States.

Index

%

%Abs meta-SQL function 2-6
%AllowNotification variable 3-1
%AllowRecipientLookup variable 3-1
%Appserver meta-HTML 4-3
%AppsRel meta-HTML 4-3
%AsOfDate variable 3-2
%AuthenticationToken variable 3-2
%BINARYSORT meta-SQL function 2-6
%BPName variable 3-2
%Browser meta-HTML 4-3
%BrowserPlatform meta-HTML 4-3
%BrowserVersion meta-HTML 4-4
%ClientDate variable 3-2
%ClientTimeZone variable 3-2
%Cols meta-HTML 4-4
%CompIntfcName variable 3-3
%Component meta-HTML 4-4
%Component variable 3-3
%Concat meta-SQL function 2-7
%ContentID variable 3-3
%ContentType variable 3-3
%Copyright meta-HTML 4-4
%Copyright variable 3-4
%Currency variable 3-4
%CurrentDateIn meta-SQL function 2-8
%CurrentDateOut meta-SQL function 2-8
%CurrentDateTimeIn meta-SQL function 2-8
%CurrentDateTimeOut meta-SQL function 2-8
%CurrentTimeIn meta-SQL function 2-8
%CurrentTimeOut meta-SQL function 2-8
%Date variable 3-4
%DateAdd meta-SQL function 2-9
%DateDiff meta-SQL function 2-9
%DateIn meta-SQL function 2-9
%DateNull meta-SQL function 2-10
%DateOut meta-SQL function 2-10
%DatePart meta-SQL function 2-11
%DateTime variable 3-4
%DateTimeDiff meta-SQL function 2-11
%DateTimeIn meta-SQL function 2-12
%DateTimeNull meta-SQL function 2-12
%DateTimeOut meta-SQL function 2-13
%DBName meta-HTML 4-4
%DbName variable 3-4
%DbServerName variable 3-5
%DBType meta-HTML 4-5
%DbType variable 3-5
%DecDiv meta-SQL function 2-13
%DecMult meta-SQL function 2-14
%DTTM meta-SQL function 2-14
%EffDtCheck meta-SQL function 2-15
%EmailAddress variable 3-5
%EmployeeId variable 3-5
%Encode meta-HTML 4-5
%ExternalAuthInfo variable 3-5
%Formname meta-HTML 4-5
%HPTabName variable 3-6
%Image meta-HTML 4-5
%Import variable 3-6
%InsertSelect meta-SQL function 2-18
%InsertValues meta-SQL function 2-20
%IsMultiLanguageEnabled variable 3-6
%JavaScript meta-HTML 4-6
%Join meta-SQL function 2-21
%KeyEqual meta-SQL function 2-23
%KeyEqualNoEffDt meta-SQL function 2-24
%LabelTag meta-HTML 4-6
%Language variable 3-6
%Language_Base variable 3-6
%Like meta-SQL function 2-25
%LikeExact meta-SQL function 2-27
%List meta-SQL function 2-29
%ListBind meta-SQL function 2-32
%ListEqual meta-SQL function 2-33
%LocalNode variable 3-6
%Market variable 3-7
%MaxInterlinkSize variable 3-8
%MaxMessageSize variable 3-8
%Menu meta-HTML 4-6
%Menu variable 3-8
%Message meta-HTML 4-7
%Mode variable 3-8
%NavigatorHomePermissionList variable 3-8
%Node variable 3-9
%NoUppercase meta-SQL function 2-34
%OldKeyEqual meta-SQL function 2-35
%OPRCLAUSE meta-SQL function 2-36
%Page meta-HTML 4-8
%Page variable 3-10
%PasswordExpired variable 3-11
%PermissionLists variable 3-11
%Portal variable 3-11
%PrimaryPermissionList variable 3-11
%ProcessProfilePermissionList variable 3-11
%PSAuthResult variable 3-12
%Request variable 3-12
%Response variable 3-12
%ResultDocument variable 3-12
%Roles variable 3-13
%Round meta-SQL function 2-36

%RowSecurityPermissionList variable 3-13
%RunningInPortal variable 3-13
%ServerTimeZone variable 3-13
%ServicePack meta-HTML 4-8
%Session variable 3-13
%SignonUserId variable 3-13
%SignonUserPswd variable 3-14
%SMTPBlackberryReplyTo variable 3-14
%SMTPGuaranteed variable 3-14
%SMTPSender variable 3-15
%SQL meta-SQL function 2-37
%SQLRows variable 3-15
%SubmitScriptName meta-HTML 4-8
%SUBREC meta-SQL function 2-39
%Substring meta-SQL function 2-39
%SyncServer variable 3-15
%Table meta-SQL function 2-40
%Test meta-SQL function 2-41
%TextIn meta-SQL function 2-41
%Time variable 3-15
%TimeAdd meta-SQL function 2-42
%TimeIn meta-SQL function 2-43
%TimeNull meta-SQL function 2-43
%TimeOut meta-SQL function 2-44
%TimePart meta-SQL function 2-44
%ToolsRel meta-HTML 4-8
%TransformData variable 3-16
%TrimSubstr meta-SQL function 2-44
%Truncate meta-SQL function 2-45
%Truncatetable meta-SQL function 2-45
%UpdatePairs meta-SQL function 2-46
%Upper meta-SQL function 2-47
%URL meta-HTML 4-8
%UserDescription variable 3-16
%UserId meta-HTML 4-9
%UserId variable 3-16
%WLInstanceID variable 3-16
%WLName variable 3-16

A

Abs function 1-19
accessing
  AESection 5-2
  Component Interface structure 10-61
  PSMessages collection 27-14
AccruableDays function 1-19
AccruableFactor function 1-20
Acos function 1-21
ActiveRowCount function 1-22
AddAttachment function 1-23
AddDropDownItem function 1-28
AddEmailAddress function 1-30
AddKeyListItem function 1-30
AddSystemPauseTimes function 1-31
AddToDate function 1-33
AddToDateTime function 1-34
AddToTime function 1-35
AESection class
  accessing 5-2
  declaring 5-5
  example 5-4
  GetAESection function 1-238
  methods 5-5
  property 5-11
  scope of 5-5
All function 1-36
AllOrNone function 1-37
AllowEmplidChg function 1-38
Amortize function 1-39
API functions
  GetSession 1-278
  RemoteCall 1-374
  SendMail 1-408
application classes
  Class function 6-18
  class names 6-3
  CollectGarbage function 1-66
  constructors 6-8
  declaring 6-18
  declaring external functions 6-10
  declaring external interfaces 6-5
  declaring private 6-5
  defining methods 6-7
  downcasting 6-16
  extending 6-4
  Get function 6-20
  import declarations 6-11
  Import function 6-21
  Method function 6-22
  overriding properties 6-14
  scope of 6-18
  self-reference 6-12
  Set function 6-23
  structure 6-2
  superclass reference 6-15
  understanding 6-1
  using 6-2
Application Engine
  CallAppEngine function 1-52
  CommitWork function 1-67
  file class example 13-41
  GetAESection function 1-238
  SetTempTableInstance function 1-446
  SQL object considerations 29-9
  using with Business Interlink 8-9
Array class
  CreateArray function 1-92
  CreateArrayAny function 1-93
  CreateArrayRept function 1-94
  creating arrays 7-1
  creating empty arrays 7-5
  declaring 7-9
  fetch results 29-5
  flattening and promotion 7-8

methods 7-10
multi-dimensional arrays 7-5
populating an array 7-3
properties 7-26
removing items 7-4
scope of 7-9
Split function 1-456
SQL bind values 29-5
Array functions
CreateArray 1-92
CreateArrayAny 1-93
CreateArrayRept 1-94
Split 1-456
Asin function 1-40
AssignedPagelet
class properties 20-109
collection methods 20-111
collection property 20-114
Atan function 1-41
attachment functions
AddAttachment 1-23
CopyAttachments 1-87
DeleteAttachment 1-138
filenames for PutAttachment 1-364
GetAttachment 1-239
PutAttachment 1-363
ViewAttachment 1-506
AttributeValue
class properties 20-89
collection methods 20-91
collection property 20-94
AvailableCategory
collection methods 20-115
collection property 20-117
AvailablePagelet
class properties 20-118
collection methods 20-120
collection property 20-121

B

BIDocs methods 8-55
BIDocs properties 8-65
BlackScholesCall function 1-41
BlackScholesPut function 1-42
BootstrapYTMs function 1-43
branch collection
method 30-16
properties 30-17
Break 1-44
bulk functions
BulkDeleteField 1-44
BulkInsertField 1-47
BulkModifyPageFieldOrder 1-50
BulkDeleteField function 1-44
BulkInsertField function 1-47
BulkModifyPageFieldOrder function 1-50
Business Interlink class
%MaxInterlinkSize variable 3-8
BIDocs methods 8-55
BIDocs properties 8-65
configuration parameters 8-70
declaring 8-9
GetBiDoc function 1-242
GetInterlink function 1-257
methods 8-10
property 8-70
scope of 8-10
state of object 8-9
using 8-2
using with Application Engine 8-9

C

CallAppEngine function 1-52
Canceling functions
DoCancel 1-151
Exit 1-178
WinEscape 1-516
CancelPubHeaderXmlDoc function 1-55
CancelPubXmlDoc function 1-56
CancelSubXmlDoc function 1-57
catch section 11-3
ChangeEmailAddress function 1-58
Char function 1-59
Chart class
appearance properties 9-24
axes methods 9-32
axes properties 9-33
color methods 9-29
component processor 9-12
creating a chart 9-43
creating a Gantt chart 9-45
data methods 9-15
data properties 9-23
data type 9-14
examples 9-43
Gantt chart 9-45
GetChart function 1-245
GetChartURL function 1-246
iScript example 9-48
iScripts 9-13
legend method 9-37
legend properties 9-38
overlay methods 9-40
overlay properties 9-42
scope of 9-14
style sheets 9-11
terms 9-3
title properties 9-35
types 9-4
understanding 9-1
Chart class functions
GetChart 1-245

GetChartURL 1-246
ChartField functions
RenameDBField 1-379
RenamePage 1-380
RenameRecord 1-382
SetDBFieldAuxFlag 1-413
SetDBFieldCharDefn 1-414
SetDBFieldFormat 1-416
SetDBFieldFormatLength 1-418
SetDBFieldLabel 1-420
SetDBFieldLength 1-421
SetDBFieldNotUsed 1-423
SetPageFieldPageFieldName 1-437
SetRecFieldEditTable 1-439
SetRecFieldKey 1-441
CharType function 1-59
cheat sheets 32-1
CI data collection
methods 10-53
standard properties 10-60
class constructors
application classes 6-8
notifications classes 18-3
Class function 6-18
Clean function 1-62
ClearDropDownList function 1-63
ClearKeyList function 1-63
ClearSearchDefault function 1-64
ClearSearchEdit function 1-65
Code function 1-65
Codeb function 1-66
CollectGarbage function 1-66
CommitWork function 1-67
CompareLikeFields function 1-70
CompIntfPropInfoCollection collection methods 10-64
CompIntfPropInfoCollection collection properties 10-65
CompIntfPropInfoCollection object properties 10-66
component buffer functions
ActiveRowCount 1-22
AddKeyListItem 1-30
CompareLikeFields 1-70
ComponentChanged 1-72
CopyFields 1-88
CopyRow 1-90
CurrentLevelNumber 1-118
CurrentRowNumber 1-118
DeleteRecord 1-143
DeleteRow 1-144
DiscardRow 1-149
ExpandBindVar 1-180
FetchValue 1-184
FieldChanged 1-186
GetNextNumber 1-265
GetNextNumberWithGaps 1-267
GetRelField 1-275
GetSetID 1-279
InsertRow 1-306
PriorValue 1-358
RecordChanged 1-368
RecordDeleted 1-370
RecordNew 1-372
RowFlush 1-394
RowScrollSelect 1-395
RowScrollSelectNew 1-397
ScrollFlush 1-401
ScrollSelect 1-402
ScrollSelectNew 1-406
SetControlValue 1-425
SetCursorPos 1-427
SetDefault 1-429
SetDefaultAll 1-430
SetDefaultNext 1-430
SetDefaultNextRel 1-431
SetDefaultPrior 1-431
SetDefaultPriorRel 1-432
SetNextPage 1-436
SetTempTableInstance 1-446
SortScroll 1-455
StopFetching 1-464
TotalRowCount 1-479
TreeDetailInNode 1-492
UpDateSysVersion 1-503
UpdateValue 1-503
component buffer meta-SQL functions
%TruncateTable 2-45
component buffer system variables
%Component 3-3
%Menu 3-8
%Mode 3-8
Component Interface class
%CompIntfcName variable 3-3
accessing structure 10-61
CompIntfPropInfoCollection collection methods 10-64
CompIntfPropInfoCollection collection properties 10-65
CompIntfPropInfoCollection object properties 10-66
Component Interface collection methods 10-32
Component Interface collection property 10-33
data collection methods 10-53
data collection standard properties 10-60
data collections 10-18
data item standard property 10-61
declaring 10-7
effective dated data 10-28
GetMethodNames function 1-261
implementing 10-8
keys 10-4
life cycle 10-2
quick reference 32-25
reusing existing code 10-29
scope of 10-8

session class methods 10-31
setting keys 10-4
standard methods 10-34
standard properties 10-6, 10-50
timeouts 10-17
traversing 10-18
using data collection 10-18
Component Interface class function
StartWork 1-463
Component statement 1-71
ComponentChanged function 1-72
constructors 6-8
ContainsCharType function 1-73
ContainsOnlyCharType function 1-75
content reference
class methods 20-73
class properties 20-74
collection methods 20-85
collection property 20-88
Conversion functions
Char 1-59
Code 1-65
Codeb 1-66
ConvertChar 1-77
NumberToString 1-346
String 1-467
Value 1-506
ConvertChar function 1-77
ConvertCurrency function 1-81
ConvertDatetimeToBase function 1-83
ConvertRate function 1-84
ConvertTimeToBase function 1-86
Cookie class properties 15-28
CopyAttachments function 1-87
CopyFields function 1-88
CopyRow function 1-90
Cos function 1-90
Cot function 1-91
CreateArray function 1-92
CreateArrayAny function 1-93
CreateArrayRept function 1-94
CreateDirectory function 1-95
CreateException function 1-97
CreateJavaArray function 1-98
CreateJavaObject function 1-99
CreateMessage function 1-100
CreateObject function 1-101
CreateObjectArray function 1-104
CreateProcessRequest function 1-106
CreateRecord function 1-106
CreateRowset function 1-108
CreateSOAPDoc function 1-110
CreateSQL function 1-111
CreateXmlDoc function 1-113
creating
arrays 7-1
web libraries 15-3
CubicSpline function 1-114
CurrEffDt function 1-116
CurrEffRowNum function 1-116
CurrEffSeq function 1-117
Currency and financial functions
%Currency 3-4
Amortize 1-39
ConvertCurrency 1-81
RoundCurrency 1-394
SinglePaymentPV 1-454
UniformSeriesPV 1-502
Current date and time functions
%CurrentDateIn 2-8
%CurrentDateOut 2-8
%CurrentDateTimeIn 2-8
%CurrentDateTimeOut 2-8
%CurrentTimeIn 2-8
%CurrentTimeOut 2-8
CurrentLevelNumber function 1-118
CurrentRowNumber function 1-118
Custom display format functions
GetStoredFormat 1-283
SetDisplayFormat 1-432

D

Data Buffer Access Functions
CreateRecord 1-106
CreateRowset 1-108
GetField 1-248
GetLevel0 1-260
GetRecord 1-273
GetRow 1-276
GetRowset 1-277
Date and time functions
%DateAdd 2-9
%DateDiff 2-9
%DateIn 2-9
%DateNull 2-10
%DateOut 2-10
%DatePart 2-11
%DateTimeDiff 2-11
%DateTimeIn 2-12
%DateTimeNull 2-12
%DateTimeOut 2-13
%DTTM 2-14
%TimeAdd 2-42
%TimeIn 2-43
%TimeNull 2-43
%TimeOut 2-44
%TimePart 2-44
AddToDate 1-33
AddToDateTime 1-34
AddToTime 1-35
ConvertDatetimeToBase 1-83
ConvertTimeToBase 1-86
Date 1-120
Date3 1-120

DatePart 1-121
DateTime6 1-121
DateTimeToHTTP 1-122
DateTimeToLocalizedString 1-124
DateTimeToTimeZone 1-126
DateTimeValue 1-127
DateValue 1-128
Day 1-130
Days 1-130
Days360 1-131
Days365 1-131
FormatDateTime 1-197
GetCalendarDate 1-243
Hour 1-302
IsDate 1-310
IsDateTime 1-311
IsDaylightSavings 1-312
IsTime 1-320
meta-SQL considerations 2-1
Minute 1-336
Month 1-337
Second 1-407
Time 1-475
Time3 1-475
TimePart 1-476
TimeToTimeZone 1-477
TimeValue 1-478
TimeZoneOffset 1-478
Weekday 1-515
Year 1-523
Date function 1-120
Date3 function 1-120
DatePart function 1-121
DateTime6 function 1-121
DateTimeToHTTP function 1-122
DateTimeToLocalizedString function 1-124
DateTimeToTimeZone function 1-126
DateTimeValue function 1-127
DateValue function 1-128
Day function 1-130
Days function 1-130
Days360 function 1-131
Days365 function 1-131
DBCSTrim function 1-132
DBPatternMatch function 1-132
debugging
Java 16-11
Declare Function 1-133
Decrypt function 1-137
Default functions
SetDefault 1-429
SetDefaultAll 1-430
SetDefaultNext 1-430
SetDefaultNextRel 1-431
SetDefaultPrior 1-431
SetDefaultPriorRel 1-432
Degrees function 1-137
DeleteAttachment function 1-138
DeleteEmailAddress function 1-141
DeleteImage function 1-142
DeleteRecord function 1-143
DeleteRow function 1-144
DeleteSQL function 1-145
DeleteSystemPauseTimes function 1-147
deprecated functions 32-68
DisableMenuItem function 1-149
DiscardRow function 1-149
DoCancel function 1-151
DoModal function 1-151
DoModalComponent function 1-154
DoSave function 1-157
DoSaveNow function 1-158
Double-byte character functions
CharType 1-59
ContainsCharType 1-73
ContainsOnlyCharType 1-75
ConvertChar 1-77
DBCSTrim 1-132
downcasting 6-16
DynamicCategory collection
methods 20-122
property 20-125

E

Effective date functions
%EffDtCheck 2-15
CurrEffDt 1-116
CurrEffRowNum 1-116
CurrEffSeq 1-117
NextEffDt 1-341
NextRelEffDt 1-342
PriorEffDt 1-357
PriorRelEffDt 1-358
email functions
AddEmailAddress 1-30
ChangeEmailAddress 1-58
DeleteEmailAddress 1-141
MarkPrimaryEmailAddress 1-328
EnableMenuItem function 1-160
EncodeURL function 1-161
EncodeURLForQueryString function 1-161
Encrypt function 1-162
EncryptNodePswd function 1-163
EndMessage function 1-164
EndModal function 1-165
EndModalComponent function 1-166
environment functions
ExpandEnvVar 1-181
GetCwd 1-247
GetEnv 1-247
Error 1-167
EscapeHTML function 1-169
EscapeJavascriptString function 1-170
EscapeWML function 1-171

Evaluate 1-171
Exact function 1-172
exception class
CreateException function 1-97
data type 11-4
methods 11-4
properties 11-7
scope of 11-4
Throw language construct 1-474
throwing exceptions 11-2
Try language construct 1-494
try-catch sections 11-3
understanding 11-1
using 11-2
Exec function 1-173
Executable files
Exec function 1-173
WinExec function 1-517
ExecuteRolePeopleCode function 1-176
ExecuteRoleQuery function 1-177
ExecuteRoleWorkflowQuery function 1-178
Existence functions
All 1-36
AllOrNone 1-37
None 1-345
OnlyOne 1-353
OnlyOneOrNone 1-354
Exit 1-178
exiting loops 1-44
Exp function 1-179
ExpandBindVar function 1-180
ExpandEnvVar function 1-181
ExpandSqlBinds function 1-181

F

Fact function 1-182
faults
SOAPDoc class 28-7
favorite
class properties 20-154
collection methods 20-155
collection property 20-159
FetchSQL function 1-183
FetchValue function 1-184
field class
declaring 12-2
GetField function 1-248
methods 12-2
properties 12-9
scope of 12-2
shortcut considerations 12-1
FieldChanged function 1-186
file attachment
AddAttachment function 1-23
CopyAttachment function 1-87
DeleteAttachment function 1-138
GetAttachment function 1-239
PutAttachment filenames 1-364
PutAttachment function 1-363
ViewAttachment function 1-506
file class
access interruptions 13-3
Application Engine example 13-41
CreateDirectory function 1-95
error processing 13-6
examples 13-31
file rowset considerations 13-41
FileExists function 1-187
FindFiles function 1-193
GetFile function 1-249
methods 13-8
multiple file layouts 13-44
plain text files 13-4
properties 13-28
reading multiple file layouts 13-45
ReadRecord example 13-35
ReadRowset example 13-40
RemoveDirectory function 1-377
scope of 13-2
security considerations 13-3
WriteRecord example 13-32
WriteRowset example 13-37
writing multiple file layouts 13-48
FileExists function 1-187
Financial functions
AccruableDays 1-19
AccruableFactor 1-20
BlackScholesCall 1-41
BlackScholesPut 1-42
BootstrapYTMs 1-43
ConvertRate 1-84
CubicSpline 1-114
HermiteCubic 1-296
HistVolatility 1-301
LinearInterp 1-324
Find function 1-189
Findb function 1-190
FindCodeSetValues function 1-191
FindFiles function 1-193
flattening arrays 7-8
FlushBulkInserts function 1-195
folder
class method 20-63
class properties 20-64
collection methods 20-69
collection property 20-73
For 1-196
FormatDateTime function 1-197
Function statement 1-198
functions
declaring 1-133
declaring for application classes 6-10
defining 1-198
return values 1-388

G

Gantt chart example 9-45
GenerateActGuideContentUrl function 1-200
GenerateActGuidePortalUrl function 1-201
GenerateActGuideRelativeUrl function 1-203
GenerateaHomepageRelativeURL function 1-218
GenerateComponentContentRelURL function 1-204
GenerateComponentContentURL function 1-206
GenerateComponentPortalRelURL function 1-209
GenerateComponentPortalURL function 1-211
GenerateComponentRelativeURL function 1-214
GenerateExternalPortalURL function 1-216
GenerateExternalRelativeURL function 1-217
GenerateHomepageRelativeURL function 1-219
GenerateQueryContentURL function 1-220
GenerateQueryPortalURL function 1-222
GenerateQueryRelativeURL function 1-224
GenerateScriptContentRelURL function 1-226
GenerateScriptContentURL function 1-227
GenerateScriptPortalRelURL function 1-229
GenerateScriptPortalURL function 1-231
GenerateScriptRelativeURL function 1-232
GenerateTree function 1-234
GenerateWorklistPortalURL function 1-235
GenerateWorklistRelativeURL function 1-236
Get function 6-20
GetAESection function 1-238
GetAttachment function 1-239
GetBiDoc function 1-242
GetCalendarDate function 1-243
GetChart function 1-245
GetChartURL function 1-246
GetCwd function 1-247
GetEnv function 1-247
GetField function 1-248
GetFile function 1-249
GetGrid function 1-253
GetHTMLText function 1-255
GetImageExtents function 1-256
GetInterlink function 1-257
GetJavaClass function 1-259
GetLevel0 function 1-260
GetMessage function 1-262
GetMessageInstance function 1-263
GetMessageXmlDoc function 1-263
GetNextNumber function 1-265
GetNextNumberWithGaps function 1-267
GetNextProcessInstance function 1-268
GetNRXmlDoc function 1-269
GetPage function 1-270
GetPubContractInstance function 1-271
GetPubHeaderXmlDoc function 1-272
GetPubXmlDoc function 1-273
GetRecord function 1-273
GetRelField function 1-275
GetRow function 1-276
GetRowset function 1-277
GetSession function 1-278
GetSetID function 1-279
GetSQL function 1-280
GetStoredFormat function 1-283
GetSubMessage function 1-284
GetSubXmlDoc function 1-286
GetSyncLogData function 1-287
GetURL function 1-289
GetUserOption function 1-290
GetWLFieldValue function 1-290
Global statement 1-291
Gray function 1-292
grid class
  changing a grid name 1-253
  declaring 14-3
  GetGrid function 1-253
  gridcolumn 14-6
  method 14-4
  properties 14-6
  scope of 14-3
  shortcut considerations 14-2
  specifying grid column name 14-4
  using in PeopleCode 14-2
gridcolumn class
  properties 14-7

H

Hash function 1-295
having criteria 22-14
HermiteCubic function 1-296
Hide function 1-296
HideMenuItem function 1-298
HideRow function 1-298
HideScroll function 1-300
HistVolatility function 1-301
Hour function 1-302
HTML
  EscapeHTML function 1-169
  GetHTMLText function 1-255
HTML variables
  %Appserver 4-3
  %AppsREl 4-3
  %Browser 4-3
  %BrowserPlatform 4-3
  %BrowserVersion 4-4
  %Cols 4-4
  %Component 4-4
  %Copyright 4-4
  %DBName 4-4
  %DBType 4-5
  %Encode 4-5
  %Formname 4-5
  %Image 4-5

%JavaScript 4-6
%LabelTag 4-6
%Menu 4-6
%Message 4-7
%Page 4-8
%ServicePack 4-8
%SubmitScriptName 4-8
%ToolsRel 4-8
%URL 4-8
%UserId 4-9

I

Idiv function 1-303
If 1-303
image functions
DeleteImage 1-142
GetImageExtents 1-256
InsertImage 1-304
implementing
Component Interface objects 10-8
tree classes 30-8
Tree Viewer 33-3
import declarations 6-11
Import function 6-21
Import manager
%Import system variable 3-6
importing
application classes 6-11
notification classes 18-2
InsertImage function 1-304
InsertRow function 1-306
Int function 1-307
Integer function 1-308
Integration Broker functions
EncryptNodePswd 1-163
FindCodeSetValues 1-191
NodeDelete 1-342
NodeRename 1-344
NodeSaveAs 1-344
Transform 1-491
internet functions
%ContentID variable 3-3
%ContentType variable 3-3
%EmailAddress variable 3-5
%HPTabName variable 3-6
%LocalNode variable 3-6
%Node variable 3-9
%Portal variable 3-11
%Request variable 3-12
%Response variable 3-12
%RunningInPortal variable 3-13
%UserDescription variable 3-16
AddEmailAddress 1-30
ChangeEmailAddress 1-58
CreateSOAPDoc 1-110
DeleteEmailAddress 1-141
EncodeURL 1-161
EncodeURLForQueryString 1-161
EscapeHTML 1-169
EscapeJavascriptString 1-170
EscapeWML 1-171
GenerateActGuideContentUrl 1-200
GenerateActGuidePortalUrl 1-201
GenerateActGuideRelativeUrl 1-203
GenerateaHomepageRelativeURL 1-218
GenerateComponentContentRelURL 1-204
GenerateComponentContentURL 1-206
GenerateComponentPortalRelURL 1-209
GenerateComponentPortalURL 1-211
GenerateComponentRelativeURL 1-214
GenerateExternalPortalURL 1-216
GenerateExternalRelativeURL 1-217
GenerateHomepageRelativeURL 1-219
GenerateQueryContentURL 1-220
GenerateQueryPortalURL 1-222
GenerateQueryRelativeURL 1-224
GenerateScriptContentRelURL 1-226
GenerateScriptContentURL 1-227
GenerateScriptPortalRelURL 1-229
GenerateScriptPortalURL 1-231
GenerateScriptRelativeURL 1-232
GenerateTree 1-234
GenerateWorklistPortalURL 1-235
GenerateWorklistRelativeURL 1-236
GetChartURL 1-246
GetHTMLText 1-255
GetMethodNames 1-261
GetURL 1-289
MarkPrimaryEmailAddress 1-328
Unencode 1-496
ViewContentURL 1-511
ViewURL 1-512
Internet Script
chart example 9-48
charts 9-13
creating 15-3
Internet Script classes
accessing 15-7
Cookie class 15-12
Cookie class properties 15-28
creating web libraries 15-3
data types 15-9
generate functions 15-7
overview 15-1
quick reference 32-63
Request class 15-11
Request class methods 15-12
Request class properties 15-15
Response class 15-11
Response class methods 15-21
scope of 15-9
security 15-5
URI 15-3
URL 15-3

when to use 15-5
IsAlpha Function 1-309
IsAlphaNumeric Function 1-309
iScript *See* Internet Script classes
IsDate Function 1-310
IsDateTime Function 1-311
IsDaylightSavings function 1-312
IsDigits Function 1-313
IsHidden function 1-313
IsMenuItemAuthorized function 1-314
IsMessageActive function 1-316
IsModal function 1-317
IsModalComponent function 1-318
IsNumber Function 1-319
IsSearchDialog function 1-320
IsTime Function 1-320
IsUserInPermissionList function 1-321
IsUserInRole function 1-322

J

Java class
considerations using PeopleCode 16-10
CreateJavaArray function 1-98
CreateJavaObject function 1-99
data type 16-12
error handling 16-11
from Java to PeopleCode 16-7
from PeopleCode to Java 16-4
GetJavaClass function 1-259
mapping PeopleCode to Java data types 16-9
naming classes and packages 16-2
scope of 16-12
state management concerns 16-4
supported versions of Java 16-2
using the Java debugging environment 16-11
using your own classes 16-2

L

Language constructs
Break 1-44
Component 1-71
Declare function 1-133
Evaluate 1-171
Exit 1-178
For 1-196
Function 1-198
Global 1-291
If 1-303
Local 1-325
Repeat 1-383
Return 1-388
Throw 1-474
Try 1-494
While 1-515
Language preferences
%IsMultiLanguageEnabled variable 3-6
%Language variable 3-6
%Language_Base variable 3-6
SetLanguage function 1-434
leaf class
methods 30-18
properties 30-24
Left function 1-322
Len function 1-323
Lenb function 1-323
level
class methods 30-31
class properties 30-32
collection methods 30-28
collection properties 30-30
LinearInterp function 1-324
Ln function 1-324
Local statement 1-325
Log10 function 1-326
loops
exiting 1-44
Lower function 1-326
LTrim function 1-327

M

mapping
functions to methods and properties 32-65
old names to new names 32-67
MarkPrimaryEmailAddress function 1-328
MarkWLItemWorked function 1-328
Math functions
%Abs meta-SQL function 2-6
Acos 1-21
Asin 1-40
Atan 1-41
Cos 1-90
Cot 1-91
Degrees 1-137
IsNumber 1-319
Max 1-329
Min 1-336
Radians 1-367
Sin 1-454
Tan 1-473
Max Function 1-329
Menu functions
DisableMenuItem 1-149
EnableMenuItem 1-160
HideMenuItem 1-298
UnGray 1-496
message class
considerations with character fields 17-7
considerations with partial records 17-6
considerations with rowset 17-5
CreateMessage function 1-100

declaring 17-2
DeleteSystemPauseTimes function 1-147
error handling 17-7
GetMessage function 1-262
GetMessageInstance function 1-263
GetPubContractInstance function 1-271
GetSubMessage function 1-284
GetSyncLogData function 1-287
IsMessageActive function 1-316
methods 17-8
populating 17-3
properties 17-29
ReturnToServer function 1-388
scope of 17-3
Message display functions
EndMessage 1-164
Error 1-167
MessageBox 1-330
MsgGet 1-338
MsgGetExplainText 1-339
MsgGetText 1-340
Prompt 1-359
Warning 1-513
WinMessage 1-519
MessageBox function 1-330
messaging
%MaxMessageSize variable 3-8
AddSystemPauseTimes function 1-31
CancelPubHeaderXmlDoc 1-55
CancelPubXmlDoc 1-56
CancelSubXmlDoc 1-57
CreateMessage function 1-100
CreateXmlDoc function 1-113
DeleteSystemPauseTimes function 1-147
GetMessage function 1-262
GetMessageInstance function 1-263
GetMessageXmlDoc function 1-263
GetNRXmlDoc function 1-269
GetPubContractInstance function 1-271
GetPubHeaderXmlDoc 1-272
GetPubXmlDoc 1-273
GetSubMessage function 1-284
GetSubXmlDoc 1-286
GetSyncLogData function 1-287
IsMessageActive function 1-316
message class 17-1
PingNode function 1-356
PublishXmlDoc function 1-362
ReturnToServer function 1-388
SetChannelStatus function 1-424
SOAP 28-2
SOAPDoc components 28-2
SyncRequestXmlDoc function 1-472
Transform function 1-491
using SOAPDoc objects 28-8
meta-HTML
%Appserver 4-3
%AppsRel 4-3
%Browser 4-3
%BrowserPlatform 4-3
%BrowserVersion 4-4
%Cols 4-4
%Component 4-4
%Copyright 4-4
%DBName 4-4
%DBType 4-5
%Encode 4-5
%Formname 4-5
%Image 4-5
%JavaScript 4-6
%LabelTag 4-6
%Menu 4-6
%Message 4-7
%Page 4-8
%ServicePack 4-8
%SubmitScriptName 4-8
%ToolsRel 4-8
%URL 4-8
%UserId 4-9
comments 4-2
functions 4-2
understanding 4-1
variables 4-2
Meta-SQL functions 2-1
%Abs 2-6
%BINARYSORT 2-6
%Concat 2-7
%CurrentDateIn 2-8
%CurrentDateOut 2-8
%CurrentDateTimeIn 2-8
%CurrentDateTimeOut 2-8
%CurrentTimeIn 2-8
%CurrentTimeOut 2-8
%DateAdd 2-9
%DateDiff 2-9
%DateIn 2-9
%DateNull 2-10
%DateOut 2-10
%DatePart 2-11
%DateTimeDiff 2-11
%DateTimeIn 2-12
%DateTimeNull 2-12
%DateTimeOut 2-13
%DecDiv 2-13
%DecMult 2-14
%DTTM 2-14
%FirstRows 2-16
%OPRCLAUSE 2-36
%Round 2-36
%SUBREC 2-39
%Substring 2-39
%Test 2-41
%TimeIn 2-43
%TimeNull 2-43
%TimeOut 2-44
%TimePart 2-44

%TrimSubstr 2-44
%Truncate 2-45
%TruncateTable 2-45
date considerations 2-1
placement considerations 2-3
shortcuts 2-48
Method function 6-22
methods
defining for application classes 6-7
Min Function 1-336
Minute function 1-336
Mod function 1-337
Modal component functions
DoModalComponent 1-154
EndModalComponent 1-166
IsModalComponent 1-318
Month function 1-337
MsgGet function 1-338
MsgGetExplainText function 1-339
MsgGetText function 1-340

N

NextEffDt function 1-341
NextRelEffDt function 1-342
node class (PortalRegistry API)
properties 20-48
node class (Tree API)
methods 30-34
properties 30-47
NodeDelete function 1-342
NodeRename function 1-344
NodeSaveAs function 1-344
None function 1-345
Notification class
method 18-7
properties 18-8
notification classes
constructors 18-3
creating 18-3
creating a WorklistEntry 18-25
examples 18-25
importing 18-2
notification class method 18-7
notification class properties 18-8
NotificationAddress class properties 18-12
NotificationTemplate class methods 18-13
NotificationTemplate class properties 18-16
scope of 18-1, 18-2
updating a WorklistEntry 18-27
WorklistEntry class methods 18-18
WorklistEntry class properties 18-21
NotificationAddress
class properties 18-12
NotificationTemplate
class methods 18-13
class properties 18-16
NumberToString Function 1-346
Numeric functions
%DecDiv 2-13
%DecMult 2-14
%Round 2-36
%Truncate 2-45
Abs 1-19
Acos 1-21
Asin 1-40
Cos 1-90
Cot 1-91
Degress 1-137
Exp 1-179
Fact 1-182
Idiv 1-303
Int 1-307
Integer 1-308
Ln 1-324
Log10 1-326
Mod 1-337
Product 1-359
Radians 1-367
Rand 1-367
Round 1-393
Sign 1-453
Sin 1-454
Sqrt 1-463
Tan 1-473
Truncate 1-494

O

object functions
CreateObject 1-101
CreateObjectArray 1-104
ObjectDoMethod 1-349
ObjectDoMethodArray 1-350
ObjectGetProperty 1-351
ObjectSetProperty 1-352
ObjectDoMethod function 1-349
ObjectDoMethodArray function 1-350
ObjectGetProperty function 1-351
ObjectSetProperty function 1-352
OnlyOne function 1-353
OnlyOneOrNone function 1-354

P

page appearance functions
AddDropDownItem 1-28
ClearDropDownList 1-63
GetPage 1-270
Gray 1-292
Hide 1-296
HideRow 1-298
HideScroll 1-300

IsHidden 1-313
SetLabel 1-433
Unhide 1-498
UnhideRow 1-500
UnhideScroll 1-501
page class
data type 19-2
GetPage function 1-270
properties 19-2
scope of 19-2
shortcut considerations 19-1
pagelet
class properties 20-134
collection methods 20-139
collection property 20-142
PageletCategory
class method 20-125
class properties 20-126
collection methods 20-130
collection property 20-133
PeopleBooks
printed, ordering iii
PeopleCode
from Java 16-7
Java class considerations 16-10
mapping to Java data types 16-9
to Java 16-4
PeopleCode classes
AESection 5-1
application 6-1
array 7-1
business interlink 8-1
chart 9-1
Component Interface 10-1
exception 11-1
field 12-1
file 13-1
grid 14-1
gridcolumn 14-6
Internet Script 15-1
Java 16-1
message 17-1
notification 18-1
page 19-1
PortalRegistry 20-1
ProcessRequest 21-1
query 22-1
quick reference 32-1
record 23-1
row 24-1
rowset 25-1
search 26-1
session 27-1
SOAPDoc 28-1
SQL 29-1
tree 30-1
XmlDoc 31-1
PeopleCode syntax
quick reference 32-68
PermissionValue
class properties 20-95
collection methods 20-96
collection property 20-99
personalization functions
GetUserOption 1-290
SetUserOption 1-453
PingNode function 1-356
portal
class method 20-55
class properties 20-56
collection methods 20-56
collection properties 20-58
portal registry classes
AssignedPagelet class properties 20-109
AssignedPagelet collection methods 20-111
AssignedPagelet collection property 20-114
AttributeValue collection methods 20-91
AttributeValue collection property 20-94
AttributeValue properties 20-89
AvailableCategory collection methods 20-115
AvailableCategory collection property 20-117
AvailablePagelet class properties 20-118
AvailablePagelet collection methods 20-120
AvailablePagelet collection property 20-121
content reference collection methods 20-85
content reference collection property 20-88
content reference methods 20-73
content reference properties 20-74
content references 20-14
data type 20-20
deleting content considerations 20-19
DynamicCategory collection methods 20-122
DynamicCategory collection property 20-125
error handling 20-9
examples 20-159
favorite class properties 20-154
favorite collection methods 20-155
favorite collection property 20-159
folder collection methods 20-69
folder collection property 20-73
folder method 20-63
folder properties 20-64
hierarchy 20-12
life-cycle 20-11
methods 20-25
node class properties 20-48
node collection methods 20-51
node collection property 20-52
node template collection properties 20-62
NodeTemplate class methods 20-59
NodeTemplate collection methods 20-60
overview 20-1
pagelet collection methods 20-139
pagelet collection property 20-142
pagelet properties 20-134
PageletCategory collection methods 20-130

PageletCategory collection property 20-133
PageletCategory method 20-125
PageletCategory properties 20-126
PermissionValue collection methods 20-96
PermissionValue collection property 20-99
PermissionValue properties 20-95
portal class method 20-55
portal class properties 20-56
portal collection methods 20-56
portal collection properties 20-58
portal registry collection methods 20-45
portal registry collection property 20-47
properties 20-42
quick reference 32-32
RemoteNode collection methods 20-53
RemoteNode collection property 20-54
saving content considerations 20-19
scope of 20-21
search class methods 26-8
security considerations 20-5
SelectedPagelet class properties 20-149
SelectedPagelet collection methods 20-151
SelectedPagelet collection property 20-153
session class methods 20-21
TabDefinition collection methods 20-106
TabDefinition collection property 20-109
TabDefinition properties 20-100
UserHomepage Class method 20-143
UserHomepage Class properties 20-144
UserTab Class properties 20-145
UserTab collection methods 20-146
UserTab collection property 20-149
using 20-5
using Cascading Permissions 20-7
using PermissionValue 20-7
ValidFrom and ValidTo 20-9
PriorEffDt function 1-357
PriorRelEffDt function 1-358
PriorValue function 1-358
private
instance variables 6-5
methods 6-5
Process Scheduler functions
CreateProcessRequest 1-106
GetNextProcessInstance 1-268
SetupScheduleDefnItem 1-452
ProcessRequest class
CreateProcessRequest function 1-106
data type 21-2
examples 21-31
format values 21-5
GetNextProcessInstance function 1-268
item options 21-2
methods 21-8
output type values 21-5
properties 21-22
scope of 21-2
SetupScheduleDefnItem function 1-452
Product function 1-359
promotion in arrays 7-8
Prompt function 1-359
Proper function 1-361
property overrides 6-14
PSMessage class
properties 27-17
PSMessages collection
accessing 27-14
methods 27-14
PublishXmlDoc function 1-362
PutAttachment function 1-363

Q

Query
class methods 22-27
class properties 22-41
collection methods 22-24
collection property 22-27
query classes 22-1
adding criteria 22-142
collections in 22-1
creating a new query 22-140
data type 22-15
error handling 22-13
examples 22-140
having criteria 22-14
hierarchy 22-3
life-cycle 22-2
overview 22-5
Query class methods 22-27
query class properties 22-41
query collection 22-24
Query collection methods 22-24
query collection property 22-27
query fields 22-14
query metadata class properties 22-115
query metadata collection methods 22-112
query metadata collection property 22-114
QueryCriteria class methods 22-83
QueryCriteria class properties 22-88
QueryCriteria collection methods 22-81
QueryCriteria collection property 22-83
QueryDBRecord class methods 22-123
QueryDBRecord class properties 22-125
QueryDBRecord collection methods 22-121
QueryDBRecord collection property 22-123
QueryDBRecordField class properties 22-129
QueryDBRecordField collection methods 22-126
QueryDBRecordField collection property 22-129
QueryExpression class properties 22-101
QueryExpression collection methods 22-99
QueryExpression collection property 22-101
QueryField class properties 22-74

QueryField collection methods 22-69
QueryField collection property 22-72
QueryField methods 22-73
QueryList methods 22-105
QueryList property 22-107
QueryListValue properties 22-107
QueryOutputFields 22-14
QueryPrompt class properties 22-136
QueryPrompt collection methods 22-133
QueryPrompt collection property 22-135
QueryRecord class method 22-66
QueryRecord class properties 22-67
QueryRecord collection methods 22-64
QueryRecord collection property 22-66
QueryRecordHierarchy class properties 22-111
QueryRecordHierarchy collection methods 22-108
QueryRecordHierarchy collection property 22-110
QuerySecurityProfile class methods 22-117
QuerySecurityProfile class properties 22-117
QuerySelect class methods 22-49
QuerySelect class properties 22-57
QuerySelect collection methods 22-60
QuerySelect collection property 22-63
QuerySelectedFields 22-14
QueryStatistics properties 22-116
quick reference 32-45
running a query 22-11
scope of 22-15
security 22-12
session methods 22-16
using query metadata 22-10
working with criteria and expressions 22-7
QueryCriteria
adding example 22-142
class methods 22-83
class properties 22-88
collection methods 22-81
collection property 22-83
QueryDBRecord
class methods 22-123
class properties 22-125
collection methods 22-121
collection property 22-123
QueryDBRecordField
class properties 22-129
collection methods 22-126
collection property 22-129
QueryExpression
class properties 22-101
collection methods 22-99
collection property 22-101
QueryField
class methods 22-73
class properties 22-74
collection methods 22-69
collection property 22-72
QueryList
class methods 22-105
class property 22-107
QueryListValue
class properties 22-107
QueryPrompt
class properties 22-136
collection methods 22-133
collection property 22-135
QueryRecord
class method 22-66
class properties 22-67
collection methods 22-64
collection property 22-66
QueryRecordHierarchy
class properties 22-111
collection methods 22-108
collection property 22-110
QuerySecurityProfile
class methods 22-117
class properties 22-117
QuerySelect
class methods 22-49
class properties 22-57
collection methods 22-60
collection property 22-63
QueryStatistics
class properties 22-116

R

Radians function 1-367
Rand function 1-367
record class
data type 23-3
GetRecord function 1-273
methods 23-3
properties 23-23
scope of 23-3
shortcut considerations 23-2
SQL methods 23-2
RecordChanged function 1-368
RecordDeleted function 1-370
RecordNew function 1-372
referencing
application classes 6-12
superclasses 6-15
regional settings object
about 27-4
properties 27-21
RemoteCall
function 1-374
RemoveDirectory function 1-377
RenameDBField function 1-379
RenamePage function 1-380
RenameRecord function 1-382
Repeat 1-383

Replace function 1-383
Rept function 1-384
Request class
methods 15-12
properties 15-15
Response class methods 15-21
ReSubmitPubHeaderXmlDoc function 1-385
ReSubmitPubXmlDoc function 1-386
ReSubmitSubXmlDoc function 1-387
Return 1-388
ReturnToServer function 1-388
ReValidateNRXmlDoc function 1-390
RevalidatePassword function 1-391
Right function 1-392
Round function 1-393
RoundCurrency function 1-394
row class
data type 24-2
GetRow function 1-276
methods 24-2
properties 24-8
scope of 24-2
shortcut considerations 24-2
RowFlush function 1-394
RowScrollSelect function 1-395
RowScrollSelectNew function 1-397
rowset class
data type 25-2
GetRowset function 1-277
methods 25-3
properties 25-27
scope of 25-2
shortcut considerations 25-2
RTrim function 1-399
Running executable files
Exec function 1-173
WinExec function 1-517

S

Saving functions
DoSave 1-157
DoSaveNow 1-158
ScrollFlush function 1-401
ScrollSelect function 1-402
ScrollSelect functions
RowScrollSelect 1-395
RowScrollSelectNew 1-397
ScrollFlush 1-401
ScrollSelect 1-402
ScrollSelectNew 1-406
SortScroll 1-455
ScrollSelectNew function 1-406
search classes
data type 26-6
error handling 26-5
examples 26-46
hierarchy 26-4
overview 26-1
PortalRegistry class methods 26-8
quick reference 32-28
scope of 26-6
Search FS option class properties 26-40
Search HTTP option class properties 26-37
Search language collection methods 26-29
Search start options class properties 26-45
Search start options collection methods 26-42
Search start options collection property 26-44
SearchField class properties 26-16
SearchField collection methods 26-15
SearchField collection property 26-16
SearchIndex class method 26-19
SearchIndex class properties 26-20
SearchIndexes collection methods 26-17
SearchIndexes collection property 26-18
SearchLanguage class properties 26-32
SearchLanguage collection property 26-31
SearchQuery class methods 26-10
SearchQuery class properties 26-10
SearchRecord option class properties 26-22
SearchRecordField class properties 26-28
SearchRecordField collection methods 26-25
SearchRecordField collection property 26-27
SearchResult class properties 26-14
SearchResult collection methods 26-12
SearchResult collection property 26-13
SearchSchedule class properties 26-35
SearchSchedule collection methods 26-32
SearchSchedule collection property 26-35
session class methods 26-6
Search FS option class
properties 26-40
Search functions
AddKeyListItem 1-30
ClearKeyList 1-63
ClearSearchDefault 1-64
ClearSearchEdit 1-65
IsSearchDialog 1-320
SetSearchDefault 1-443
SetSearchDialogBehavior 1-444
SetSearchEdit 1-445
Search HTTP option class
properties 26-37
Search language
collection methods 26-29
search query *See* search classes
Search start options
class properties 26-45
collection methods 26-42
collection property 26-44
SearchField
class properties 26-16
collection methods 26-15
collection property 26-16
SearchIndex

class method 26-19
class properties 26-20
SearchIndexes
collection methods 26-17
collection property 26-18
SearchLanguage
class properties 26-32
collection property 26-31
SearchQuery
class method 26-10
class properties 26-10
SearchRecord
option class properties 26-22
SearchRecordField
class properties 26-28
collection methods 26-25
collection property 26-27
SearchResult
class properties 26-14
collection methods 26-12
collection property 26-13
understanding 26-3
using collections 26-2
SearchSchedule
class properties 26-35
collection methods 26-32
collection property 26-35
Second function 1-407
Secondary page functions
DoModal 1-151
EndModal 1-165
IsModal 1-317
security
%AuthenticationToken variable 3-2
%EmployeeId variable 3-5
%ExternalAuthInfo variable 3-5
%NavigatorHomePermissionList variable 3-8
%OPRCLAUSE meta-SQL 2-36
%PasswordExpired variable 3-11
%PermissionLists variable 3-11
%PrimaryPermissionList variable 3-11
%ProcessProfilePermissionList variable 3-11
%PSAuthResult variable 3-12
%ResultDocument variable 3-12
%Roles variable 3-13
%RowSecurityPermissionList variable 3-13
%SignonUserId variable 3-13
%SignonUserPswd variable 3-14
%UserId variable 3-16
AllowEmplidChg function 1-38
Decrypt function 1-137
Encrypt function 1-162
ExecuteRolePeopleCode function 1-176
ExecuteRoleQuery function 1-177
ExecuteRoleWorkflowQuery function 1-178
Hash function 1-295
Internet Scripts 15-5
IsMenuItemAuthorized function 1-314
IsUserInPermissionList function 1-321
IsUserInRole function 1-322
portal registry classes 20-5
query classes 22-12
RevalidatePassword function 1-391
SetAuthenticationResult function 1-412
SetPasswordExpired function 1-438
SwitchUser function 1-470
SelectedPagelet
class properties 20-149
collection methods 20-151
collection property 20-153
self-reference *See* application classes, self-reference
SendMail function 1-408
session class
access to PeopleSoft system 27-2
Component Interface methods 10-31
declaring 27-6
error handling 27-3
GetSession function 1-278
methods 27-7
portal registry methods 20-21
properties 27-11
PSMessages class properties 27-17
PSMessages collection 27-14
PSMessages collection methods 27-14
query class methods 22-16
quick reference 32-23
regional settings overview 27-4
regional settings properties 27-21
scope of 27-7
search method 26-6
security 27-2
trace settings overview 27-4
trace settings properties 27-24
tree classes methods 30-11
Set function 6-23
SetAuthenticationResult function 1-412
SetChannelStatus function 1-424
SetControlValue function 1-425
SetCursorPos function 1-427
SetDBFieldAuxFlag function 1-413
SetDBFieldCharDefn function 1-414
SetDBFieldFormat function 1-416
SetDBFieldFormatLength function 1-418
SetDBFieldLabel function 1-420
SetDBFieldLength function 1-421
SetDBFieldNotUsed function 1-423
SetDefault function 1-429
SetDefaultAll function 1-430
SetDefaultNext function 1-430
SetDefaultNextRel function 1-431
SetDefaultPrior function 1-431
SetDefaultPriorRel function 1-432
SetLabel function 1-433
SetLanguage function 1-434
SetNextPage function 1-436

SetPageFieldPageFieldName function 1-437
SetPasswordExpired function 1-438
SetRecFieldEditTable function 1-439
SetRecFieldKey function 1-441
SetReEdit function 1-442
SetSearchDefault function 1-443
SetSearchDialogBehavior function 1-444
SetSearchEdit function 1-445
SetTempTableInstance function 1-446
SetTracePC function 1-447
SetTraceSQL function 1-450
SetupScheduleDefnItem function 1-452
SetUserOption function 1-453
shortcut considerations
field class 12-1
grid class 14-2
page class 19-1
record class 23-2
row class 24-2
rowset class 25-2
Sign function 1-453
Sin function 1-454
SinglePaymentPV function 1-454
SOAPDoc class
accessing sections 28-6
body section 28-5
CreateSOAPDoc function 1-110
creating 28-4, 28-20
data type 28-8
envelope section 28-5
examples 28-20
fault section 28-7
header section 28-4
message exchange mode 28-2
method section 28-5
methods 28-9
overview 28-1
parts of 28-2
properties 28-17
reading 28-21
required PeopleCode 28-6
scope of 28-8
using 28-21
validating 28-7
XmlDoc 28-4
SortScroll function 1-455
Split function 1-456
SQL
%FirstRows meta-SQL function 2-16
%InsertSelect meta-SQL function 2-18
%InsertValues meta-SQL function 2-20
%Join meta-SQL function 2-21
%KeyEqual meta-SQL function 2-23
%KeyEqualNoEffDt meta-SQL function 2-24
%Like meta-SQL function 2-25
%LikeExact meta-SQL function 2-27
%List meta-SQL function 2-29
%ListBind meta-SQL function 2-32
%ListEqual meta-SQL function 2-33
%NoUppercase meta-SQL function 2-34
%OldKeyEqual meta-SQL function 2-35
%SQL meta-SQL function 2-37
%SQLRows variable 3-15
%Table meta-SQL function 2-40
%UpdatePairs meta-SQL function 2-46
binding 29-3
CreateSQL function 1-111
DeleteSQL function 1-145
executing 29-3
ExpandSqlBinds function 1-181
FetchSQL function 1-183
FlushBulkInserts function 1-195
GetSQL function 1-280
Meta functions 2-1
reusing a cursor 29-6
SetTempTableInstance function 1-446
SQLExec function 1-457
StoreSQL function 1-465
SQL class
application engine considerations 29-9
array of any 29-5
binding and executing statements 29-3
CreateSQL function 1-111
data type 29-9
DeleteSQL function 1-145
fetching 29-4
FetchSQL function 1-183
GetSQL function 1-280
global declaration considerations 29-9
methods 29-10
properties 29-16
record methods 29-3
reusing a cursor 29-6
scope of 29-9
SELECT styles 29-5
SQL definition 29-3
SQLExec function 1-457
StoreSQL function 1-465
SQL date and time functions
%DateIn 2-9
%DateOut 2-10
%DateTimeIn 2-12
%DateTimeOut 2-13
%DTTM 2-14
%TimeIn 2-43
%TimeOut 2-44
%TimePart 2-44
SQL definition
creating 29-3
SQL shortcuts
%Delete 2-49
%Insert 2-49
%SelectAll 2-49
%SelectByKey 2-51
%SelectByKeyEffDt 2-51
%SelectDistinct 2-50

%Update 2-51
SQLExec function 1-457
Sqrt function 1-463
StartWork function 1-463
StopFetching function 1-464
StoreSQL function 1-465
String function 1-467
String functions
%Concat 2-7
%Substring 2-39
%TextIn 2-41
%TrimSubstr 2-44
%Upper 2-47
Clean 1-62
Code 1-65
Codeb 1-66
DBCSTrim 1-132
DBPatternMatch 1-132
Exact 1-172
ExpandBindVar 1-180
Find 1-189
Findb 1-190
IsAlpha 1-309
IsAlphaNumeric 1-309
IsDigits 1-313
Left 1-322
Len 1-323
Lenb 1-323
Lower 1-326
LTrim 1-327
NumberToString 1-346
Proper 1-361
Replace 1-383
Rept 1-384
Right 1-392
RTrim 1-399
String 1-467
Substitute 1-468
SubString 1-469
Substringb 1-469
Upper 1-505
Value 1-506
Subrecords
%SUBREC meta-SQL function 2-39
Substitute function 1-468
SubString function 1-469
Substringb function 1-469
SwitchUser function 1-470
SyncRequestXmlDoc function 1-472
System variables 3-1
%AllowNotification 3-1
%AllowRecipientLookup 3-1
%AsOfDate 3-2
%AuthenticationToken 3-2
%BPName 3-2
%ClientDate 3-2
%ClientTimeZone 3-2
%CompIntfcName 3-3
%Component 3-3
%ContentID 3-3
%ContentType 3-3
%Copyright 3-4
%Currency 3-4
%Date 3-4
%DateTime 3-4
%DbName 3-4
%DbServerName 3-5
%DbType 3-5
%EmailAddress 3-5
%EmployeeId 3-5
%ExternalAuthInfo 3-5
%HPTabName 3-6
%Import 3-6
%IsMultiLanguageEnabled 3-6
%Language 3-6
%Language_Base 3-6
%LocalNode 3-6
%Market 3-7
%MaxInterlinkSize 3-8
%MaxMessageSize 3-8
%Menu 3-8
%Mode 3-8
%NavigatorHomePermissionList 3-8
%Node 3-9
%OperatorClass 3-9
%OperatorId 3-9
%OperatorRowLevelSecurityClass 3-9
%Page 3-10
%Panel 3-10
%PanelGroup 3-10
%PasswordExpired 3-11
%PermissionLists 3-11
%Portal 3-11
%PrimaryPermissionList 3-11
%ProcessProfilePermissionList 3-11
%PSAuthResult 3-12
%Request 3-12
%Response 3-12
%ResultDocument 3-12
%Roles 3-13
%RowSecurityPermissionList 3-13
%RunningInPortal 3-13
%ServerTimeZone 3-13
%Session 3-13
%SignonUserId 3-13
%SignonUserPswd 3-14
%SMTPBlackberryReplyTo 3-14
%SMTPGuaranteed 3-14
%SMTPSender 3-15
%SQLRows 3-15
%SyncServer 3-15
%Time 3-15
%TransformData 3-16
%UserDescription 3-16
%UserId 3-16
%WLInstanceID 3-16

%WLName 3-16

T

TabDefinition
class properties 20-100
collection methods 20-106
collection property 20-109
Tan function 1-473
Throw language construct 1-474
Time and date functions
%DateAdd 2-9
%DateDiff 2-9
%DateIn 2-9
%DateNull 2-10
%DateOut 2-10
%DatePart 2-11
%DateTimeDiff 2-11
%DateTimeIn 2-12
%DateTimeNull 2-12
%DateTimeOut 2-13
%DTTM 2-14
%TimeAdd 2-42
%TimeIn 2-43
%TimeNull 2-43
%TimeOut 2-44
%TimePart 2-44
AddToDate 1-33
AddToDateTime 1-34
AddToTime 1-35
ConvertDatetimeToBase 1-83
ConvertTimeToBase 1-86
Date 1-120
Date3 1-120
DatePart 1-121
DateTime6 1-121
DateTimeToHTTP 1-122
DateTimeToLocalizedString 1-124
DateTimeToTimeZone 1-126
DateTimeValue 1-127
DateValue 1-128
Day 1-130
Days 1-130
Days360 1-131
Days365 1-131
FormatDateTime 1-197
GetCalendarDate 1-243
Hour 1-302
IsDate 1-310
IsDateTime 1-311
IsDaylightSavings 1-312
IsTime 1-320
Minute 1-336
Month 1-337
Second 1-407
Time 1-475
Time3 1-475
TimePart 1-476
TimeToTimeZone 1-477
TimeValue 1-478
TimeZoneOffset 1-478
Weekday 1-515
Year 1-523
Time function 1-475
Time3 function 1-475
TimePart function 1-476
TimeToTimeZone function 1-477
TimeValue function 1-478
TimeZoneOffset function 1-478
TotalRowCount function 1-479
Trace functions
SetTracePC 1-447
SetTraceSQL 1-450
trace settings object
about 27-4
properties 27-24
Transfer function 1-480
Transfer functions
AddKeyListItem 1-30
ClearKeyList 1-63
DoModalComponent 1-154
EndModalComponent 1-166
IsModalComponent 1-318
SetNextPage 1-436
Transfer 1-480
TransferNode 1-484
TransferPage 1-489
TransferPortal 1-487
TransferNode function 1-484
TransferPage function 1-489
TransferPortal function 1-487
Transform function 1-491
tree
class methods 30-55
class properties 30-77
collection method 30-92
collection properties 30-93
tree classes
branch collection methods 30-16
branch collection properties 30-17
branch collections 30-15
collections in 30-4
declaring 30-7
error handling 30-5
implementing 30-8
insertion verification 30-7
leaf 30-17
leaf methods 30-18
leaf properties 30-24
level 30-31
level collection 30-28
level collection methods 30-28
level methods 30-31
level properties 30-32
node class 30-34

node methods 30-34
node properties 30-47
quick reference 32-56
relationship between 30-2
scope of 30-7
session class methods 30-11
tree 30-54
tree collection 30-91
tree collection method 30-92
tree collection properties 30-93
tree methods 30-55
tree properties 30-77
tree structure 30-94
tree structure collection 30-109
tree structure collection method 30-110
tree structure collection properties 30-111
tree structure methods 30-95
tree structure properties 30-98
TreeDetailInNode function 1-492
using tree viewer 33-1
tree structure
class methods 30-95
class properties 30-98
collection method 30-110
collection properties 30-111
Tree Viewer
invoking 33-3
multi-node example 33-7
no multi-node example 33-6
overview 33-1
TreeDetailInNode function 1-492
TriggerBusinessEvent function 1-493
Truncate function 1-494
Try language construct 1-494
try sections 11-3
type checking functions
IsAlpha 1-309
IsAlphaNumeric 1-309
IsDate 1-310
IsDateTime 1-311
IsDigits 1-313
IsNumber 1-319
IsTime 1-320
Max 1-329
Min 1-336
NumberToString 1-346

U

Unencode function 1-496
UnGray function 1-496
UnHide function 1-498
UnhideRow function 1-500
UnhideScroll function 1-501
UniformSeriesPV function 1-502
UpDateSysVersion function 1-503
UpdateValue function 1-503
UpdateXmlDoc function 1-504
Upper function 1-505
URL vs. URI 15-3
User Security functions
AllowEmplidChg 1-38
Decrypt 1-137
EncodeURL 1-162
ExecuteRolePeopleCode 1-176
ExecuteRoleQuery 1-177
ExecuteRoleWorkflowQuery 1-178
Hash 1-295
IsMenuItemAuthorized 1-314
IsUserInPermissionList 1-321
IsUserInRole 1-322
RevalidatePassword 1-391
SetAuthenticationResult 1-412
SetPasswordExpired 1-438
SwitchUser 1-470
UserHomepage
class method 20-143
class properties 20-144
UserTab
class properties 20-145
collection methods 20-146
collection property 20-149
using
application classes 6-2
content references 20-14
exceptions 11-2

V

validation functions
Error 1-167
RevalidatePassword 1-391
SetCursorPos 1-427
SetReEdit 1-442
Warning 1-513
Value function 1-506
variables
declaring 1-71, 1-291, 1-325
ViewAttachment function 1-506
ViewContentURL function 1-511
viewing trees 33-1
ViewURL function 1-512

W

Warning 1-513
web libraries
creating 15-3
GetMethodNames function 1-261
security 15-5
Weekday function 1-515
While 1-515
WinEscape function 1-516

WinExec function 1-517
WinMessage function 1-519
Workflow
notification classes 18-1
Workflow functions
GenerateActGuideContentUrl 1-200
GenerateActGuidePortalUrl 1-201
GenerateActGuideRelativeUrl 1-203
GenerateWorklistPortalURL 1-235
GenerateWorklistRelativeURL 1-236
GetWLFieldValue 1-290
MarkWLItemWorked 1-328
TriggerBusinessEvent 1-493
Workflow system variables
%AllowNotification 3-1
%AllowRecipientLookup 3-1
%BPName 3-2
%SMTPBlackberryReplyTo 3-14
%SMTPGuaranteed 3-14
%SMTPSender 3-15
Worklist system variables
%WLInstanceID 3-16
%WLName 3-16
WorklistEntry
class methods 18-18
class properties 18-21
creating 18-25
updating 18-27

X

XmlDoc class
considerations 31-3
CreateSOAPDoc function 1-110
creating 31-3
data type 31-5
error handling 31-4
methods 31-6
property 31-17
scope of 31-5
SOAPDoc considerations 28-4, 31-5
using 31-2
when to use 31-2
XmlDoc functions
CreateSOAPDoc 1-110
CreateXmlDoc 1-113
GetMessageXmlDoc 1-263
GetNRXmlDoc 1-269
PublishXmlDoc 1-362
ReSubmitPubHeaderXmlDoc 1-385
ReSubmitPubXmlDoc 1-386
ReSubmitSubXmlDoc 1-387
ReValidateNRXmlDoc 1-390
SyncRequestXmlDoc 1-472
Transform 1-491
UpdateXmlDoc 1-504
XmlNode class
data type 31-5
methods 31-17
properties 31-38
scope of 31-5

Y

Year function 1-523