

# Programming Guide

*iPlanet™ Unified Development Server*

**Version 5.0**

August 2001

Copyright (c) 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of the product or of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Forte, iPlanet, Unified Development Server, and the iPlanet logo are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Federal Acquisitions: Commercial Software - Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

---

Copyright (c) 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, California 94303, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. a les droits de propriété intellectuels relatants à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et sans la limitation, ces droits de propriété intellectuels peuvent inclure un ou plus des brevets américains énumérés à <http://www.sun.com/patents> et un ou les brevets plus supplémentaires ou les applications de brevet en attente dans les Etats - Unis et dans les autres pays.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, Forte, iPlanet, Unified Development Server, et le logo iPlanet sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.

# Contents

<b>List of Figures</b> .....	<b>23</b>
<b>List of Tables</b> .....	<b>27</b>
<b>List of Procedures</b> .....	<b>29</b>
<b>List of Code Examples</b> .....	<b>33</b>
<b>Preface</b> .....	<b>37</b>
Product Name Change .....	37
Audience for This Guide .....	38
Organization of This Guide .....	38
Text Conventions .....	41
<b>Other Documentation Resources</b> .....	<b>42</b>
iPlanet UDS Documentation .....	43
Express Documentation .....	43
WebEnterprise and WebEnterprise Designer Documentation .....	43
Online Help .....	44
<b>iPlanet UDS Example Programs</b> .....	<b>44</b>
<b>Viewing and Searching PDF Files</b> .....	<b>44</b>
<b>Chapter 1 How to Structure a Graphical User Interface</b> .....	<b>47</b>
<b>About iPlanet UDS Windows</b> .....	<b>47</b>
Inherited Windows .....	49
Named Event Handlers and Inherited Windows .....	50
Nested Windows .....	51
Named Event Handlers and Nested Windows .....	53
Inherited Windows or Nested Windows? .....	54
Windows as Page Templates .....	54
Page Formatting .....	55

<b>Structuring the User Interface</b> .....	<b>57</b>
Opening and Closing Windows .....	57
Writing a Display Method .....	58
Creating the Window Object .....	59
Displaying the Window .....	59
Blocking the Calling Window .....	60
Displaying a Nested Window .....	61
Using Dialog Boxes .....	63
Writing the Window Event Loop .....	64
Using Event Handlers .....	65
Child Events .....	65
Input Focus Event Chain .....	65
Field Selection Event Chain .....	67
Close Window Event Chain .....	68
Event Loop for Inherited Windows .....	68
Event Loop Using a Nested Window .....	69
<b>Chapter 2 Using Complex Widgets</b> .....	<b>71</b>
<b>About Tab Folders</b> .....	<b>71</b>
Creating Tab Folders in the Window Workshop .....	73
Using the New > TabFolder Command .....	73
Using the Group Into > TabFolder Command .....	74
Editing the Tab Folder .....	76
Setting Tab Folder Properties .....	79
Creating Tab Folders Dynamically .....	81
<b>About Outline, List View, and Tree View Fields</b> .....	<b>84</b>
<b>Using Outline Fields</b> .....	<b>88</b>
Interacting with Outline Fields .....	88
Data for Outline Fields .....	88
Node Hierarchy .....	89
Event Handling .....	90
Outline Field Properties .....	90
Providing Controls .....	90
Displaying the Root Node .....	90
Turning Scrollbars On and Off .....	90
Controlling Row Highlights and Scroll Policy .....	90
Displaying Column Titles .....	91
Individual Column Properties .....	91
Column Content .....	91
Column State .....	92
Column Sizing and Alignment .....	92
Column Indenting .....	92
Creating an Outline Field in the Window Workshop .....	93

<b>Using Outline Fields</b> ( <i>continued</i> )	
Providing Data for an Outline Field	96
Using a Subclass of DisplayNode	97
Creating and Assigning the Node Hierarchy	98
<b>Using List View Fields</b>	<b>102</b>
List View Styles	102
Styles and Portability	103
Styles and Sorting	103
Interacting with List View Fields	104
Data for List View Fields	104
List View Properties	105
List View Style	105
Scrollbars	105
Row Highlights and Scroll Policy	106
Column Titles	106
Individual Column Properties	107
Column Content	107
Column State	107
Column Sizing and Alignment	107
Creating a List View Field in the Window Workshop	108
Setting List View Style	108
Column Names and Other Column Properties	108
Providing Data for a List View Field	112
Small Icon and Simple List Fields	112
Image List View Field	112
Detail List View Field	112
Using a DisplayNode Array	112
Using a Subclass of DisplayNode	113
Creating and Assigning the Node Array	114
<b>Using Tree View Fields</b>	<b>115</b>
Interacting with Tree View Fields	116
Data for Tree View Fields	116
DVSmallIcon and DVSelectedIcon Attributes	117
DVNodeText Attribute	117
Node Hierarchy	117
Event Handling	118
Tree View Properties	118
Providing Controls	118
Allowing Dragging	118
Displaying the Root Node	118
Turning Scroll Bars On and Off	119
Row Highlights and Scroll Policy	119

<b>Using Tree View Fields</b> ( <i>continued</i> )	
Creating a Tree View Field in the Window Workshop	120
Providing Data for a Tree View Field	121
Attributes for Positioning Nodes	122
<b>Chapter 3 Creating a Portable User Interface</b>	<b>127</b>
<b>Designing a Portable User Interface</b>	<b>127</b>
Widget Differences	128
Tab Folders	128
List Views and Tree Views	128
Fonts	129
Image Resolution	130
Styles	131
Tools for Portable Displays	134
Grid Fields	134
Field Size Partnerships	135
Field Size Policies	136
Using Grid Fields	137
Nesting Grid Fields	137
Resizing Fields Within a Grid Field	138
Row and Column Alignment	140
Specifying the Window's Border	143
Using Column and Row Partnerships	145
<b>Chapter 4 Implementing Online Help</b>	<b>147</b>
<b>Overview</b>	<b>147</b>
Context-Sensitive Help	149
Default Help File	149
DefaultHelpFile Attribute	149
Float-Over Help	152
Enabling Float-Over Help	152
Providing Float-Over Help Text	153
Float-Over Help for Palette Lists	154
Suppressing Float-Over Help Text	155
Status-Line Help	155
Status-Line Help for Palette Lists	158
Status-Line Help for Menu Widgets	159
<b>Using the Prefabricated Help Commands</b>	<b>160</b>
Default Help File	162
Implementing the About Command	162

<b>Chapter 5 Testing the User Interface</b> .....	<b>163</b>
<b>Using the TestClient Utility</b> .....	<b>163</b>
Starting the TestClient Utility .....	164
Portable Syntax .....	164
OpenVMS Syntax .....	164
The TestClient Window .....	165
Setting TestClient Options .....	166
Leaving the TestClient Utility .....	166
Testing the Client .....	166
<b>Using the AutoTester Project</b> .....	<b>167</b>
Capturing Input in an Input Capture Class .....	167
Setting Up for Input Capture .....	169
Capturing Input .....	170
Dumping State Information .....	172
Making Portable File References .....	174
Playing Back Captured Input .....	175
Analyzing the Results .....	178
Automating Regression Tests .....	179
Creating Your Own Test Utility .....	180
<b>Chapter 6 Using iPlanet UDS Logging Tools</b> .....	<b>181</b>
<b>About iPlanet UDS Logging</b> .....	<b>181</b>
Logging Requirements .....	183
iPlanet UDS Logging Tools .....	184
<b>iPlanet UDS Logging Filters</b> .....	<b>185</b>
Message Types .....	185
Service Types .....	186
Shortcuts for Specifying all Service Types .....	187
iPlanet UDS Internal Service Types .....	187
Group Numbers .....	188
Level Numbers .....	188
Useful Message Filters .....	189
<b>Implementing Your Logging Scheme with LogMgr Methods</b> .....	<b>190</b>
Referencing the LogMgr Object .....	191
Logging Application Information with PutLine .....	191
Logging Message Text .....	192
Logging State Information .....	192
Put and PutLine Examples .....	192
Altering the Flow of Control of an Application .....	194
Test Method Example .....	194
Changing Logging Filters .....	195
Flushing Current Log Files .....	195
Flush .....	196

<b>Setting up Logging with iPlanet UDS</b> .....	<b>196</b>
Tools for Setting Log Flags .....	197
The FORTE_LOGGER_SETUP Environment Variable .....	197
The Log Flags Page of the iPlanet UDS Control Panel .....	197
The -fl Flag of iPlanet UDS Commands .....	198
The Utility > Modify Log Flags Command .....	198
The Component > Properties Command .....	198
The Component > Modify Log Flags Command .....	198
Modifying Log Flags with Fscript and Escript .....	199
The LogMgr.ModifyFlags Method .....	199
Order of Precedence for Log-Setting Procedures .....	200
Precedence Details on NT .....	200
Setting Log Flags with Command Line Syntax .....	201
Log Flag Syntax .....	201
FORTE_LOGGER_SETUP Example .....	203
Command-Line Log Flags Example .....	205
Setting Log Flags with a Window Interface .....	206
Specifying Message and Service Types .....	207
Specifying Group Levels .....	208
<b>Choosing a Testing Mode</b> .....	<b>209</b>
Local Testing .....	209
Distributed Testing .....	209
Distributed Testing and Test Service Object Names .....	210
<b>Locating Logging Output</b> .....	<b>210</b>
Changing Log File Names for Active Partitions .....	211
Rules for Log File Names .....	211
<b>Logging Examples</b> .....	<b>214</b>
Auction Example .....	214
Setting up Logging .....	216
Running the Example .....	218
Logging for Performance Example .....	219
Log Flags Used .....	219
Example Method That Times Itself .....	220
Example Timer Output Method .....	221
<b>Chapter 7 Deployment Concepts</b> .....	<b>223</b>
<b>An Overview of Deploying Applications and Libraries</b> .....	<b>223</b>
Distributed Applications and Application Distributions .....	224
Libraries .....	225
Environments .....	225



<b>Configuring Applications</b> .....	<b>225</b>
Client Configuration .....	226
Applets .....	226
Server Configuration .....	226
Relationship Between Partitions and Projects .....	226
Relationship Between Partitions and Libraries .....	227
Automatic Partitioning .....	227
Testing a Configuration .....	228
Deploying Applications .....	228
<b>About Partitions</b> .....	<b>228</b>
Logical Partitions .....	229
Client Partition .....	229
Server Partition .....	229
Replicated Server Partition .....	229
Router Partition .....	229
Non-replicated Server Partition .....	230
Reference Partition .....	230
<b>Configuring Libraries</b> .....	<b>233</b>
<b>About Environments</b> .....	<b>234</b>
Development Environment .....	234
Deployment Environment .....	234
Simulating Deployment Environments .....	235
Connected Environments .....	235
Nodes .....	236
<b>Chapter 8 Deploying iPlanet UDS Applications and Libraries</b> .....	<b>237</b>
<b>About Deploying Applications and Libraries</b> .....	<b>237</b>
iPlanet UDS Utilities for Deploying Applications and Libraries .....	239
Examples .....	239
Getting Started .....	240
<b>Creating a Default Application Configuration</b> .....	<b>240</b>
How the Default Configuration is Generated .....	242
How Client Partitions Are Assigned .....	242
How Server Partitions Are Assigned .....	243
Why Some Service Objects Are Unassigned .....	243
Why Service Objects Are Not Replicated .....	243
Examining the Logical Partitions .....	244
Examining Nodes in an Application Configuration .....	245
Examining the Assigned Partitions .....	246

<b>Customizing the Application Configuration</b> .....	<b>247</b>
Choosing a Simulated Environment Definition .....	249
Redefining Logical Partitions .....	249
Moving Service Objects Between Partitions .....	250
Creating a New Logical Partition .....	250
Making a Reference Partition .....	251
Using Reference Partitions with Connected Environments .....	254
Defining a Client Partition as an Applet .....	256
Combining Service Objects and Partitions .....	257
Modifying a Service Object Definition .....	260
Failover and Load Balancing .....	261
Setting the Export Name and External Type .....	262
Specifying the Environment Search Path .....	262
Changing Partition Assignments .....	264
Adding Partition Assignments .....	264
Moving Partitions .....	264
Deleting Partitions .....	265
Changing Assigned Partition Properties .....	265
Assigned Client Partition Properties .....	265
Assigned Server Partition Properties .....	266
Changing Configuration Properties .....	268
Viewing and Setting the Configuration Properties .....	268
Recreating the Default Configuration .....	269
<b>Making an Application Distribution</b> .....	<b>269</b>
Understanding Application Distributions .....	270
Standard Partitions .....	270
Compiled Partitions .....	270
Launching Applets and Other Applications .....	271
Adding an Icon File for Windows to the Distribution .....	272
Application Distribution Directory .....	272
File Naming Conventions .....	275
Using the Make Distribution Command .....	276
Local/Remote Option .....	277
Auto-Compile Option .....	277
Full or Partial Make Options .....	277
Install in Current Environment Option .....	278
The Make Distribution Command and Compiled Partitions .....	279
Environment Variables and Path .....	279
Using the fcompile Command for Compiled Partitions .....	281
Compiling a Partition for Use on Several Computing Platforms .....	283

<b>Making an Application Distribution</b> ( <i>continued</i> )	
Packaging an Application Distribution	284
Installing Additional Files with Your Application Distribution	284
Documenting a Distribution	285
<b>Installing an Application Distribution</b>	<b>286</b>
Transferring a Distribution to a Deployment Environment	287
Loading a Distribution into an Environment Repository	288
When a Distribution Conflicts with an Installed Application	291
Modifying a Partitioning Configuration	292
Reassigning Partition Assignments	292
Modifying Installed or Assigned Partition Properties	295
<b>Installing the Application</b>	<b>296</b>
Installing Applications on Server Nodes	298
Installing Applications on Client Nodes	299
Generating Icons for Standard Client Partitions	301
Generating Icons for Compiled Client Partitions	302
Creating Icons by Hand	303
Installing Applications with Reference Partitions	303
Completing Partial Installations	303
<b>Deploying a Library</b>	<b>304</b>
Creating an Library Configuration	304
Creating a Default Library Configuration	305
Examining Library Configurations	306
Examining the Projects	306
Examining Nodes in a Library Configuration	307
Examining Assigned Libraries	308
Modifying a Library Configuration	308
Adding Projects to the Configuration	309
Removing Libraries from a Node	310
Standard or Compiled Libraries	311
Making a Library Distribution	312
About Library Distributions	312
Using the Make Distribution Command	315
Compiling Libraries	315
Installing a Library Distribution	317
<b>Removing and Updating Applications and Libraries</b>	<b>319</b>
Removing an Application or Library	319
Upgrading Applications	320
Upgrading Installed Applications	320
Upgrading Reference Partitions	321
Upgrading Libraries	322
Partial Upgrades	323

<b>Chapter 9 Class Runtime Properties</b> .....	<b>325</b>
<b>Class Runtime Properties</b> .....	<b>325</b>
Class Runtime Property Defaults and Performance .....	327
Setting Runtime Properties for a Class .....	328
Setting Runtime Property Attributes for an Object .....	329
Runtime Attributes on Nested Objects .....	329
Runtime Attributes on Cloned Objects .....	330
<b>Distributed Objects</b> .....	<b>330</b>
Named and Unnamed Anchored Objects .....	332
Non-Distributed Anchored Objects .....	332
Invoking Methods on Distributed Objects .....	332
Accessing Attributes of Distributed Objects .....	333
Note on DataValue Subclasses in Framework .....	335
<b>Shared Objects</b> .....	<b>335</b>
Automatic Locking: Mutexes .....	336
Nonshared Objects and Concurrent Access .....	337
Waiting for Events and Shared Objects .....	337
Nested Method Invocations .....	339
Common Mutex Deadlock .....	339
Avoiding Common Mutex Deadlock .....	340
Distributed Mutex Deadlock .....	340
Distributed Recursive Deadlock .....	341
Distributed Shared Objects .....	342
Cloning Shared Objects .....	343
<b>Transactional Objects</b> .....	<b>343</b>
Transactional Logging .....	345
Common Transactional Logging Error .....	346
Shared Transactional Objects and Transactional Locking .....	347
Updating Non-Shared Transactional Objects .....	348
Read Locks .....	349
Write Locks .....	350
Lock Promotion .....	351
Transactional Deadlock .....	351
Lock Promotion Deadlock .....	352
Locking in Nested Transactions .....	355
Transaction Task Participants and Locking .....	355
Transactional Objects Not in Transaction .....	356
Cloning Transactional Objects .....	356
Distributed Transactions and the Transactional Property .....	356
<b>Monitored Objects</b> .....	<b>357</b>

<b>Chapter 10 Using Interfaces</b> .....	<b>359</b>
<b>About Interfaces</b> .....	<b>359</b>
Interface Elements .....	360
Implementing an Interface .....	360
Using an Interface as a Data Type .....	360
Dynamic Class Loading .....	361
Multiple Interface Inheritance .....	361
Polymorphism .....	361
Interface Hierarchies .....	362
Creating an Interface .....	362
Implementing an Interface .....	364
Implementing Multiple Interfaces .....	366
Using an Interface as a Declared Type .....	367
Interface Elements .....	368
Virtual Attributes .....	368
Methods .....	369
Events .....	370
Event Handlers .....	370
Constants .....	370
<b>Dynamic Class Loading</b> .....	<b>371</b>
Application Developer: Using Dynamic Loading within Application Code .....	372
Step 1. Defining the Interface .....	373
Step 2. Providing the Mechanism for Registering Implementing Classes .....	375
Step 3. Loading the Class and Creating the Object .....	376
Step 4. Making the Interface Library .....	380
Step 5. Testing the Application .....	381
Step 6. Delivering the Application and Interface Library .....	381
Class Implementer: Providing Implementations for Dynamic Loading .....	382
Step 1. Importing the Interface Library .....	383
Step 2. Creating the Implementation Project .....	384
Step 3. Creating the Implementing Class .....	384
Step 4. Making the Implementation Library .....	386
Step 5. Deploying the Implementation Library .....	387
Step 6. Registering the Implementation Library .....	388
Step 7. Testing the Interface .....	389
<b>Using Multiple Interface Inheritance</b> .....	<b>389</b>
Declared Type and Runtime Type for Interfaces .....	391
Example of Multiple Inheritance .....	391
Signature Conflicts .....	392

<b>Chapter 11 Working With Service Objects</b> .....	<b>393</b>
<b>About Service Objects</b> .....	<b>393</b>
TOOL Class Service Objects .....	394
Service Objects for Database Access .....	395
DBResourceMgr Service Objects .....	396
DBSession Service Objects .....	396
Setting Properties of Service Objects .....	397
Class Runtime Properties .....	398
Service Object Properties .....	398
<b>Setting Service Object Visibility</b> .....	<b>399</b>
Environment Visibility .....	400
User Visibility .....	401
<b>Assigning a Dialog Duration to Service Objects</b> .....	<b>402</b>
Dialog Duration and State Information .....	403
Dialog Duration and Error Handling .....	404
Message Duration Service Objects .....	405
State Information for Message Duration .....	406
Error Handling for Message Duration .....	406
Transactions and Message Duration .....	406
Events for Message Duration .....	407
Transaction Duration Service Objects .....	407
State Information for Transaction Duration .....	408
Error Handling for Transaction Duration .....	408
Transactions and Transaction Duration .....	409
Events for Transaction Duration .....	409
Session Duration Service Objects .....	409
State Information for Session Duration .....	410
Error Handling for Session Duration .....	410
Transactions and Session Duration .....	410
Events for Session Duration .....	411
<b>Replicating Servers for Failover and Load Balancing</b> .....	<b>411</b>
Replication for Failover .....	412
Replication for Load Balancing .....	412
Replication for Failover and Load Balancing .....	412
<b>Providing Failover</b> .....	<b>413</b>
Enabling Failover .....	413
Failover in the Local Environment .....	414
Cross-Environment Failover .....	416
Deploying Applications with Cross Environment Failover .....	417
Combining Local and Cross-Environment Failover .....	418
Relationship between Dialog Duration and Failover .....	419

<b>Providing Load Balancing</b> .....	<b>420</b>
Enabling Load Balancing .....	421
Setting the Number of Replicates .....	422
Relationship between Dialog Duration and Load Balancing .....	423
The Router Partition .....	424
Single-Threaded and Multi-Threaded Routers .....	425
Failover for the Router .....	425
<b>Sharing Service Objects Between Applications</b> .....	<b>426</b>
Sharing a Service Object in a Single Environment .....	427
Sharing a Service Object in Connected Environments .....	428
Including a Shared Service Object in an Application .....	429
Deploying the Shared Service .....	429
Making a Distribution for the Server Application in the Local Environment .....	430
Including the Supplier Project .....	430
Making the Reference Partition .....	431
<b>Using the Environment Search Path</b> .....	<b>432</b>
Specifying an Environment Search Path .....	433
Specifying Auto-Start for a Partition .....	434
<b>Chapter 12 Advanced Options for Structuring Client Applications</b> .....	<b>435</b>
<b>Writing Applications That Use the Launch Server and Applets</b> .....	<b>435</b>
Setting up the AppletSupport Library .....	436
Advantages of Using the AppletSupport Library .....	436
Restrictions .....	437
Building Applications by Starting Multiple Smaller Applications .....	437
Using the LaunchService Service Object .....	438
Using LaunchMgr Methods .....	439
A Scenario .....	439
Configuring and Deploying the Main Client Application .....	441
Testing .....	443
Customizing the Launcher Application .....	443
Deploying Applications That Launch Other Applications And Applets .....	443
Troubleshooting Client Applications That Use Applets .....	444
<b>Developing Applications with Nomadic Clients</b> .....	<b>445</b>
Connecting to the Environment .....	447
Disconnecting the Client Partition from the Distributed Environment .....	448
Releasing the Connection to Service Objects and Anchored Objects .....	448
Releasing the Connection to the Environment .....	449
Example of Connecting and Disconnecting to the Environment Manager .....	450
Starting a Nomadic Client Application .....	450
Testing Nomadic Client Applications .....	451
Restrictions on Nomadic Clients .....	452

<b>Chapter 13 Upgrading Deployed Applications</b> .....	<b>453</b>
<b>Choosing an Upgrade Approach</b> .....	<b>453</b>
Types of Upgrades .....	454
Factors Influencing Upgrade Possibilities .....	456
When is a Rolling Upgrade Necessary? .....	456
Changes Allowed Between Upgrades .....	459
About Class Versions and Compatibility Levels .....	460
About Interoperable Upgrades .....	462
Upgrading Clients .....	462
Upgrading Servers .....	463
About Compatibility Level Upgrades .....	463
About Compatibility Level Rolling Upgrades .....	464
About Class Version Upgrades .....	465
About Converters .....	467
New Method Converters .....	470
Obsolete Method Converters .....	470
Converters for Modified Methods .....	471
Event Converters .....	471
The Banking1-2 Example .....	472
<b>Performing an Interoperable Upgrade</b> .....	<b>475</b>
Summary of Upgrade Steps .....	475
Changes Allowed in an Interoperable Upgrade .....	476
Using Distributed Object References .....	476
Adding New Attributes to Objects that are not Distributed .....	477
Updating Window Classes .....	477
Making the Distribution .....	477
<b>Using Compatibility Levels to Upgrade</b> .....	<b>478</b>
Summary of Upgrade Steps .....	478
Using New Compatibility Levels of Libraries and Shared Service Objects .....	479
<b>Using Class Versions and Converters for a Rolling Upgrade</b> .....	<b>481</b>
Planning a Class Version Upgrade .....	481
Special Requirements for High Availability Servers .....	482
Summary of Upgrade Steps .....	482
Updating Classes and Writing Converters .....	483
Viewing Converters .....	483
Guidelines for Writing Converters .....	485
Writing Method Converters .....	487
Writing Event Converters .....	489
Modifying Converters .....	491
Deleting Converters .....	491
Using Class Version Numbers .....	492
Testing Converters .....	493
Making a Distribution .....	493



<b>Using Class Versions and Converters for a Rolling Upgrade</b> ( <i>continued</i> )	
Installing and Starting Updated Partitions	493
Using Failover	495
Using Load Balancing	495
Recording Information About the Update	496
Removing Versions of Classes	496
<b>Chapter 14 TOOL Reflection Classes</b>	<b>497</b>
<b>The Power of Reflection</b>	<b>497</b>
Learning About Reflection	498
Restrictions	498
<b>TOOL Class Reflection</b>	<b>499</b>
Accessing Reflection Objects	500
Getting Information About a Class or Interface	501
Accessing Arrays	503
Accessing Simple Data Types	504
Working With Attributes	504
Getting the AttributeDesc Object	504
Determining the Data Type of an Attribute	505
Getting or Setting the Attribute Value on an Object	505
Getting or Setting the Value of a Primitive Type	506
Working With Methods	508
Getting Parameter and Return Value Information	509
Invoking the Method	509
<b>Reflection Examples</b>	<b>510</b>
Object Inspector	510
Class Browser	513
<b>Chapter 15 XSLT Processor Library</b>	<b>517</b>
<b>Features of the iIS XSLT Processor</b>	<b>517</b>
Restrictions	517
<b>Introducing XML and XSL</b>	<b>518</b>
What is XML?	518
Representing an XML Document in a TOOL Application	519
What are XSL Transformations?	519
What is an XSLT Processor?	521
<b>Using an XSLT Processor in a TOOL Application</b>	<b>522</b>
Using the Results Document in a TOOL Application	523
<b>Using Protocol Handlers</b>	<b>523</b>
<b>The XSLT Processor Library Classes</b>	<b>526</b>

<b>Chapter 16 Source Code Management for iPlanet UDS Projects</b>	<b>527</b>
<b>Overview of Source Code Management</b>	<b>528</b>
<b>Source Code Management Service</b>	<b>528</b>
Features and Limitations	528
Using the GenericRepository Library	529
Using SCMServer	529
<b>Using the SourceCodeManager Library</b>	<b>531</b>
Features and Limitations	531
Creating an SCM Service	532
SCM Service Example	533
<b>Export Formats</b>	<b>533</b>
Compatibility Format	533
Multi-File Format	534
<b>Chapter 17 Performance-Based Load Balancing</b>	<b>537</b>
<b>Using Performance-Based Load Balancing</b>	<b>537</b>
Providing Performance Information About a Node	538
Using the Environment Console	538
Using Escript	539
Specifying the Number of Replicates	539
Partitioning the Replicates	541
<b>Chapter 18 Creating HTTP Applications</b>	<b>543</b>
<b>HTTP Overview</b>	<b>543</b>
Clients	544
Servers	544
The HTTPSupport Model	544
Messages	546
Headers	546
Message Body	547
Requests	548
Responses	548
Sessions	548
Application Sessions	549
Network Sessions	550
Secure Sessions	550
HTTPSupport Classes and Interfaces	551

<b>Creating HTTP Clients</b> .....	<b>553</b>
HTTP Client Requests .....	553
Specifying Request Details .....	554
SetMethod .....	554
SetURL .....	554
SetQueryString .....	554
Message Body .....	555
Entities .....	555
Message Headers .....	555
Sending Messages .....	556
Send Method .....	556
Dispatching Requests .....	556
Sessions .....	557
Reusing Sessions .....	558
<b>HTTP Servers</b> .....	<b>559</b>
Creating iPlanet UDS HTTP Server Applications .....	559
MessageReceiver Interface .....	560
HTTPReceiver Interface .....	560
Listening For Requests .....	561
Advertise Method .....	561
SetServiceEOSInfo Command .....	562
Responding to Requests .....	562
Sessions .....	563
Processing Cookies .....	563
Message Headers .....	564
Message Body .....	564
Entities .....	565
HTTPFactory .....	566
Using Multi-Threaded Server Processes .....	567
<b>Configuring HTTP Sessions</b> .....	<b>567</b>
Configuring HTTP Clients .....	567
System-Wide Configuration .....	569
Session-By-Session Configuration .....	570
Configuring HTTP Servers .....	570
HTTPConfigManager .....	572
HTTPHelper .....	572
Configuring Request Dispatching With HTTPServerManager .....	573
Configuring Secure Sessions .....	575
Secure Client Sessions .....	575
Secure Server Sessions .....	578
<b>Related Topics</b> .....	<b>579</b>
Encoding and Decoding With Base64 .....	579
Character Sets in Messages .....	579

<b>Chapter 19 Enabling Security</b> .....	<b>581</b>
<b>About SSL</b> .....	<b>581</b>
How SSL Works .....	582
<b>SSL Services</b> .....	<b>583</b>
<b>SSL Classes</b> .....	<b>584</b>
Working with Certificates .....	586
Creating a Root Certificate .....	587
Creating a Leaf Certificate .....	588
<b>Code Examples</b> .....	<b>589</b>
<b>Chapter 20 Using the XMLDOM2 Library</b> .....	<b>593</b>
<b>What Are XML Namespaces?</b> .....	<b>594</b>
<b>Tree-Based APIs</b> .....	<b>595</b>
Advantages of Using Tree-Based APIs .....	595
Restrictions When Using Tree-Based APIs .....	595
<b>The Document Object Model</b> .....	<b>595</b>
DOM Trees .....	595
DOM Tree Examples .....	596
Creating DOM Trees .....	597
Reading DOM Trees .....	598
Manipulating DOM Trees .....	598
The DOM API Classes .....	600
DOM Level 2 Features .....	601
<b>Upgrading from DOM Level 1</b> .....	<b>602</b>
<b>Chapter 21 Using the XMLSAX2 Library</b> .....	<b>603</b>
<b>XML Namespaces</b> .....	<b>604</b>
<b>SAX and the Event Handling Model</b> .....	<b>604</b>
What Are Events? .....	605
Examples of Events .....	606
StartDocument() and EndDocument() Methods .....	608
StartElement() and EndElement() Methods .....	608
Character Events .....	608
<b>Filtering Events</b> .....	<b>609</b>
The FilterImpl Class .....	609
<b>Exception Handling in SAX Level 2</b> .....	<b>610</b>
<b>New Features in SAX Level 2</b> .....	<b>610</b>
Support for Namespaces .....	610
Configurable Parsers .....	610
Other Features .....	611
<b>Upgrading from SAX Level 1</b> .....	<b>611</b>
<b>The XMLSAX2 Classes and Interfaces</b> .....	<b>611</b>

<b>Chapter 22 Accessing Internet Directory Services</b> .....	<b>615</b>
<b>LDAP Overview</b> .....	<b>615</b>
LDAP Directory Information .....	616
LDAP Directory Trees .....	616
Accessing and Updating an LDAP Directory .....	616
<b>Using the iPlanet UDS LDAP Library</b> .....	<b>617</b>
Establishing an LDAP Session .....	617
Connecting to an LDAP Server .....	618
Message IDs .....	618
Authentication .....	618
Searching an LDAP Directory .....	619
Building LDAP Filters .....	621
Updating an LDAP Directory .....	623
Adding an LDAP Entry .....	623
Modifying an Attribute for an LDAP Entry .....	625
Deleting an LDAP Entry .....	627
Closing an LDAP Session .....	628
<b>Appendix A iPlanet UDS Example Applications</b> .....	<b>629</b>
<b>How to Install iPlanet UDS Example Applications</b> .....	<b>629</b>
<b>Overview of iPlanet UDS Example Applications</b> .....	<b>630</b>
<b>Application Descriptions</b> .....	<b>631</b>
AdaptableAuction .....	632
AppletBanking .....	633
Auction .....	634
AutoTester .....	635
Banking1-2 .....	636
HTTPSupport .....	636
InheritedWindow .....	639
InternatBank .....	640
NestedWindow .....	641
NomadicOrderClient .....	642
PrintSample .....	643
TabFolders .....	644
TimeItV1-4 .....	645
TreeList .....	647
<b>Index</b> .....	<b>649</b>



# List of Figures

Figure 1-1	Example Superclass Window	49
Figure 1-2	Example Subclass Windows	49
Figure 1-3	Example Subwindow	51
Figure 1-4	Main Windows Displaying Nested Windows	52
Figure 1-5	Page Template Window	54
Figure 1-6	Printed Version of Page Template	56
Figure 2-1	Tab Folder	72
Figure 2-2	Tab Folder Components	72
Figure 2-3	Header Style Property	79
Figure 2-4	Layout Policy	80
Figure 2-5	TabFolder Properties Dialog	80
Figure 2-6	Outline Field	85
Figure 2-7	List View Field	86
Figure 2-8	Tree View Field	87
Figure 2-9	Tree View Field and List View Field Used Together	87
Figure 2-10	Data in an Outline Field	89
Figure 2-11	OutlineField Properties Dialog	96
Figure 2-12	Nodes in a Node Hierarchy	99
Figure 2-13	List View Field Styles	103
Figure 2-14	Data for List View Field	104
Figure 2-15	Data in a Tree View Field	117
Figure 2-16	Nodes in a Node Hierarchy	123
Figure 3-1	Nested Grid Fields	138
Figure 3-2	Simple Text Editor Window	139
Figure 3-3	First Grid Field	139
Figure 3-4	Second Grid Field	140
Figure 3-5	Width Policy Adjustment	140

Figure 3-6	Size Properties Dialog for the Parent Grid Field	143
Figure 3-7	Testing the Window in Figure 3-2	144
Figure 3-8	Adjusting the Border of a Window	144
Figure 5-1	TestClient Window	165
Figure 5-2	AutoTester Control Panel	170
Figure 5-3	Capture Window	171
Figure 6-1	Log Filter Syntax	203
Figure 6-2	The Log Flags Page of the Control Panel and the Modify Log Flags Dialog	206
Figure 6-3	Using a Drop List for the Message Type	207
Figure 6-4	Specifying All Service Types	208
Figure 6-5	Specifying All or a Single Group Level	208
Figure 6-6	Auction Example Starting Window	214
Figure 6-7	Paintings Listed in the Starting Window	215
Figure 6-8	Auction Example View Window	215
Figure 6-9	Auction Example Bid Window	216
Figure 7-1	Partitions in a Distributed Application	224
Figure 7-2	Relationship between Partitions and Projects	227
Figure 7-3	Reference Partition for Current Environment	231
Figure 7-4	Reference Partition for Connected Environment	232
Figure 8-1	Deploying an Application or Library	238
Figure 8-2	Logical Partition Browser	244
Figure 8-3	Logical Partitions Dialog	244
Figure 8-4	Node Properties Dialog	246
Figure 8-5	Choosing the Name of a Simulated Environment	249
Figure 8-6	Assigned Partition Properties Dialog (Client Partition)	265
Figure 8-7	Assigned Partition Properties Dialog (Server Partition)	266
Figure 8-8	Configuration Properties Dialog	269
Figure 8-9	Application Distribution Directory Structure	273
Figure 8-10	Transferring a Distribution to a Deployment Environment	287
Figure 8-11	Loading Distribution into Environment Repository	289
Figure 8-12	Installing an Application on a Server Node	299
Figure 8-13	Installing an Application on a Client Node	301
Figure 8-14	Windows Command Icon	302
Figure 8-15	Project Browser	306
Figure 8-16	Logical Partition Dialog for a Project	306
Figure 8-17	Node Properties Dialog	307
Figure 8-18	Compilation Properties for Node Dialog	308



Figure 8-19	Default Library Configuration	309
Figure 8-20	Library Distribution Directory Structure	313
Figure 9-1	Class Properties Dialog: Runtime Properties Tab Page	328
Figure 9-2	Using Distributed References for Distributed Objects	331
Figure 10-1	TaxCalculationIFace Interface	364
Figure 10-2	TaxCalculationImp Implementation of TaxCalculationIFace Interface	366
Figure 10-3	Interface Definition	374
Figure 10-4	TaxCalculationImp Implementation of TaxCalculationIFace Interface	386
Figure 10-5	Single Inheritance	389
Figure 10-6	Multiple Interface Inheritance	390
Figure 11-1	TOOL Class Service Object General Properties Tab Page	395
Figure 11-2	DBResourceMgr Service Object Database Tab Page	396
Figure 11-3	DBSession Service Object Database Tab Page	397
Figure 11-4	Environment-Visible Service Object	400
Figure 11-5	User Visible Service Object	402
Figure 11-6	Message Dialog Duration	405
Figure 11-7	Transaction Dialog Duration	408
Figure 11-8	Session Dialog Duration	410
Figure 11-9	Service Object Dialog with Failover	413
Figure 11-10	Hardware Failover	414
Figure 11-11	Software Failover	415
Figure 11-12	Cross-Environment Failover	416
Figure 11-13	Service Object Properties Environment Search Path Tab Page	417
Figure 11-14	Load Balancing on a Single Node	420
Figure 11-15	Load Balancing on Multiple Nodes	421
Figure 11-16	Load Balancing with Transaction Dialog Duration	424
Figure 11-17	Router with Failover	426
Figure 11-18	Reference Partition for Single Environment	427
Figure 11-19	Reference Partition for Connected Environment	428
Figure 12-1	Running an Application using Nomadic Clients	446
Figure 13-1	Upgrade Approach Decision Tree	458
Figure 13-2	Interoperable Upgrade with Failover	462
Figure 13-3	Compatability Level Step Upgrade	464
Figure 13-4	Compatability Level Rolling Upgrade	465
Figure 13-5	Class Version Upgrade	466
Figure 13-6	Interoperable Upgrade with Failover	468
Figure 13-7	Before Upgrade	472

Figure 13-8	Caller is Newer: New Converter Executes Caller .....	474
Figure 13-9	Caller is Older: Obsolete Converter Executes on Callee .....	475
Figure 14-1	TOOL Reflection Class Hierarchy .....	499
Figure 14-2	Using ClassType Objects to Enable Reflection .....	501
Figure 15-1	XSLT Processor Document Flow .....	521
Figure 17-1	Node Properties Dialog .....	538
Figure 17-2	Service Object Properties Dialog .....	539
Figure 17-3	Assigned Partition Properties Dialog (Server Partition) .....	541
Figure 18-1	HTTPSupport Model .....	545
Figure 19-1	SSL Class Hierarchy .....	585
Figure 20-1	Simple XML DOM Tree .....	596

# List of Tables

Table 6-1	Message Types .....	186
Table 6-2	Service Types .....	187
Table 6-3	Useful Message Filters .....	189
Table 6-4	Syntax for Specifying Log Files .....	202
Table 6-5	Platform-specific Syntax for FORTE_LOGGER_SETUP .....	204
Table 6-6	Platform-specific Location for Setting FORTE_LOGGER_SETUP .....	204
Table 6-7	Standard Output .....	210
Table 13-1	Changes Allowed During a Class Version Upgrade .....	467
Table 16-1	Classes in the GenericRepository Library .....	529
Table 16-2	Methods in the GenericRepository.SCMServer class .....	530
Table 16-3	Methods in the SourceCodeManager.SCM class .....	531
Table 16-4	Types of Files Exported in Multi-file Format .....	534
Table 16-5	Files Exported Based on Component Type .....	535
Table 18-1	Typical HTTP request message .....	546
Table 18-2	HTTPSupport Classes .....	552
Table 18-3	Typical HTTP response message from an HTTP server .....	559
Table 18-4	Configuration options for iPlanet UDS HTTP client applications .....	569
Table 18-5	Configuration options for Advertise .....	573
Table 18-6	SSL options for SetConfigValue and SetSessionValue .....	576
Table 18-7	Certificate configuration settings for secure sessions .....	577
Table 18-8	Secure Session Configuration Options for Advertise .....	578
Table 20-1	Methods for Creating Objects .....	597
Table 20-2	Methods for Fetching Objects .....	598
Table 20-3	Methods for modifying objects or their contents .....	598
Table 20-4	Methods for adding, removing, and copying nodes .....	599
Table 20-5	The DOM API classes .....	600
Table 20-6	New Exception Codes for DOM Level 2 .....	601

Table 21-1	The XMLSAX2 classes and interfaces .....	611
Table 22-1	LDAPSession.Search Parameters .....	619
Table 22-2	Modification Types .....	625

# List of Procedures

To copy the documentation to a client or server .....	45
To view and search the documentation .....	45
To display a window .....	57
To create a new tab folder .....	74
To create a tab folder .....	75
To copy an existing tab page with the Edit > Copy command .....	77
To create a new tab page .....	77
To delete a tab page .....	78
To move a tab page .....	78
To edit a tab label .....	79
To create a tab folder .....	81
To create an outline field .....	94
To use the DisplayNode subclass .....	97
To create the node hierarchy .....	99
To create a list view field .....	109
To use the DisplayNode subclass .....	113
To create a tree view field .....	120
To create the node hierarchy .....	123
To specify the DefaultHelpFile attribute .....	150
To provide context-sensitive help .....	151
To provide float-over help for a field .....	153
To provide float-over help for a palette list .....	154
To create a status line widget .....	156
To create status-line help for palette lists .....	158
To provide status-line help .....	159
To include a Help menu on a window .....	161
To use the About command .....	162

To test an application .....	166
To set up for input capture .....	169
To capture input .....	171
To prepare for playback .....	175
To play back the PencilPlay tests .....	176
To implement logging in iPlanet UDS .....	182
To edit a shortcut icon and start iPlanet UDS .....	205
To run the iPlanet UDS command with log flags .....	206
To run the iPlanet UDS command with log flags .....	206
To change a log file name for a compiled active partition or iPlanet UDS executor partition .....	212
To change the log file name for an interpreted active server partition .....	213
To test logging instrumentation on the Auction application .....	218
To create a default application configuration for client applications .....	241
To create a default application configuration for server applications .....	241
To open a configuration .....	248
To move a service object .....	250
To create a logical partition .....	250
To include the project that defines the service object .....	252
To make a reference partition .....	252
To make the distribution for the application that contains the service object .....	254
To create the reference partition with an environment search path .....	255
To create an applet .....	256
To modify a service object definition .....	260
To assign a logical partition .....	264
To delete a disabled, assigned partition .....	265
To change the configuration of an installed application without changing the contents of any logical partitions .....	278
To make a distribution for a configuration containing compiled partitions .....	279
To run fcompile for a compiled partition .....	283
To install an application distribution .....	286
To load a distribution .....	290
To reassign a partition .....	293
To copy a partition assignment .....	294
To assign an unassigned partition .....	294
To set properties of an assigned or installed partition .....	295
To install an application .....	297
To create a Windows NT client icon .....	303

To create a default library configuration .....	305
To add a project to the configuration .....	310
To remove a restricted external library .....	310
To turn on compilation for a library .....	311
To make a library distribution .....	315
To compile multiple libraries .....	316
To run fcompile for a compiled library .....	317
To deploy a library distribution .....	318
To uninstall an application or library distribution .....	319
To change the configuration of an installed application without changing the contents of any logical partitions .....	321
To upgrade an installed application .....	321
To make the reference partition reference the partition of the newer release of a changed application .....	322
To make the application reference the newer release of the library .....	323
To create an interface .....	363
To implement an interface in the Class Workshop .....	365
To use dynamic class loading with an interface .....	372
To create an interface .....	374
To load the class and create the object .....	376
To configure a library .....	380
To provide implementations for the interface .....	382
To import a library .....	383
To include a library as a supplier .....	384
To implement an interface in the Class Workshop .....	385
To configure a library .....	387
To combine local and cross-environment failover .....	418
To turn on load balancing for a service object .....	422
To specify the number of replicates for a particular configuration .....	423
To include a shared service object in an application .....	429
To make a distribution for the server application .....	430
To transfer a project from one repository to another .....	431
To configure an application as an applet .....	442
To update the Banking application .....	473
To upgrade the Banking application .....	474
To upgrade a deployed application using a new application distribution .....	476
To make the distribution for the updated, interoperable application .....	478

To upgrade a deployed application using a new application distribution .....	479
To use a new compatibility level of a library .....	480
To use a new compatibility level of a service object .....	480
To upgrade a deployed application, using class versions and converters .....	482
To see all converters defined for a class .....	483
To see the text of an individual converter .....	484
To create a method converter .....	487
To replace an event and write the associated event converters .....	490
To replace the event BugAddedOld with a new event BugAddedNew .....	490
To set a class version number .....	492
To upgrade a client partition .....	494
To upgrade a high availability server partition .....	494
To use load balancing for a server having multiple versions .....	495
To create an SCM service .....	532
To configure an iPlanet UDS HTTP client .....	568
To configure an iPlanet UDS HTTP server .....	571
To configure a secure client session .....	576
To configure a secure HTTP session .....	578
To use Adaptable Auction .....	632
To use AppletBanking .....	634
To use Auction .....	635
To use AutoTester .....	635
To use Banking1-2 .....	636
To use HTTP examples .....	637
To use InheritedWindow .....	640
To use InternatBank .....	640
To use NestedWindow .....	642
To use NomadicOrderClient .....	642
To use PrintSample .....	644
To use TabFolders .....	644
To use TimeItV1-4 .....	645
To use TimeItV1 .....	645
To use TimeItV2 .....	646
To use TimeItV3 .....	646
To use TimeItV4 .....	647
To use TreeListExample .....	648



# List of Code Examples

How to Nest a Window .....	53
A Typical Display Method .....	58
Creating a Window Object .....	59
Blocking the Calling Window .....	60
Using the start task Statement to Display a Window .....	61
Displaying a Nested Window .....	62
Using Window.MessageDialog .....	64
Registering Named Event Handlers for a Superclass .....	68
Using the register Statement .....	69
Creating a Tab Folder Dynamically .....	82
Handling the AfterTabSelect Event .....	83
Filtering Message .....	192
Logging Debugging Message .....	193
Testing LogMgr Settings .....	194
Testing for log flags and writing log output .....	217
Method that times itself .....	220
LogTime method from Code Example 6-5 .....	221
Accessing attributes of a remote object .....	333
Wrapper method to access classes that are not distributed .....	335
Using a mutex lock .....	336
Using a lock in an event loop .....	338
Using a nested lock .....	339
Example of a common mutex deadlock .....	340
Example of a distributed mutex deadlock .....	341
Distributed recursive deadlock .....	342
Marking a cloned object as a shared object .....	343
Making nested attributes transactional .....	344

Effect of transactional logging .....	345
Common transactional logging error .....	346
Transactional read lock .....	349
Transactional write lock .....	350
Transactional deadlock .....	351
Lock promotion deadlock .....	352
Two transactions in lock promotion deadlock .....	353
Setting the IsTransactional attribute for a cloned transactional object .....	356
Declaring a local variable using an interface as a data type .....	367
Using the FindLibrary and FindClass methods .....	378
Checking applet availability using the ListApplets method .....	440
Starting an applet using the RunApplet method .....	440
Shutting down applets using the Shutdown method .....	441
Connecting a client partition to the environment .....	447
Disconnecting from a service object .....	448
Disconnecting from an environment .....	449
Connecting and disconnecting a client partition .....	450
Using the GetClassType method .....	502
Accessing class information about an object .....	503
Using ArrayDesc to get information about array elements .....	503
Using PrimitiveDesc to determine the data type of a return value .....	504
Getting the data type of attributes .....	505
Using GetValue to retrieve an attribute value .....	506
Determining the primitive data type of an attribute .....	507
Getting a method's signature and invoking the method .....	508
Using the ObjectInspector class .....	510
Using the ClassBrowser class .....	514
A typical XML document generated by an application .....	518
Using XSLT to transform a document .....	520
Using the XSLT processor in an application .....	522
Implementing a custom protocol handler .....	525
Building an HTTP client request .....	553
Creating a session .....	558
Reusing a session object .....	558
Advertising a server .....	561
Processing the body of a message .....	564
Registering a user-defined entity using HTTPFactory .....	566

Registering a user-defined HTTP request .....	566
Configuring an iPlanet UDS client application .....	568
Configuring an iPlanet UDS server application .....	571
Creating a root certificate for testing SSL .....	587
Creating a leaf certificate for testing SSL .....	588
Requesting server authentication .....	589
Secure server supplying certificate for authentication .....	590
Setting up an AcceptableRoots certificate .....	591
Simple XML Sample .....	596
Simple XML Document .....	606
Output generated by parsing simple XML document .....	606
Connecting to an LDAP server .....	618
Authenticating a session with the LDAP server .....	618
Specifying an LDAP directory search .....	620
Building an “equals” filter .....	621
Building a “lexicographic greater than” filter .....	622
Building a substring filter .....	622
Building a substring filter .....	622
Building an “any attribute value” filter .....	623
Adding an entry to an LDAP directory .....	624
Modifying an LDAP directory entry .....	626
Deleting an LDAP directory entry .....	627
Unbinding from an LDAP server .....	628
Closing a session with the LDAP server .....	628
Importing iPlanet UDS examples into a repository .....	629
Removing examples from a workspace .....	630



# Preface

The *iPlanet UDS Programming Guide* provides both conceptual and how-to information about a number of topics that application developers will find useful when designing and building iPlanet UDS applications.

This preface contains the following sections:

- “Product Name Change” on page 37
- “Audience for This Guide” on page 38
- “Organization of This Guide” on page 38
- “Text Conventions” on page 41
- “Other Documentation Resources” on page 42
- “iPlanet UDS Example Programs” on page 44
- “Viewing and Searching PDF Files” on page 44

## Product Name Change

Forte 4GL has been renamed the iPlanet Unified Development Server. You will see full references to this name, as well as the abbreviations iPlanet UDS and UDS.

## Audience for This Guide

The *iPlanet UDS Programming Guide* is intended for application developers. We assume that you:

- have TOOL programming experience
- are familiar with your particular window system
- understand the basic concepts of object-oriented programming as described in *A Guide to the iPlanet UDS Workshops*
- have used the iPlanet UDS workshops to create classes

## Organization of This Guide

The following table briefly describes the contents of each chapter:

Chapter	Description
Chapter 1, “How to Structure a Graphical User Interface”	Explains how you design and display windows and how to respond to user interactions.
Chapter 2, “Using Complex Widgets”	Explains the use of iPlanet UDS’s complex widgets, including detailed information about creating the widgets in the Window Workshop, assigning data to the widgets, and manipulating the widgets dynamically in TOOL code.
Chapter 3, “Creating a Portable User Interface”	Provides information about creating a portable user interface, and how to use the portability features that iPlanet UDS provides. Topics include grid fields, field partnerships, and widget sizing properties.
Chapter 4, “Implementing Online Help”	Explains how you use built-in support for implementing an on-line help system, for providing on-line help, and for providing status-line help.

Chapter	Description
Chapter 5, “Testing the User Interface”	Provides information about testing iPlanet UDS applications and explains the use of the TestClient utility (which allows you to simulate client action) and the use of the AutoTester project (which allows you to capture user input from a test session and replay it while capturing state information).
Chapter 6, “Using iPlanet UDS Logging Tools”	Describes the use of iPlanet UDS logging tools. Logging information from other iPlanet UDS manuals is summarized here and provided with new examples and example scenarios.
Chapter 7, “Deployment Concepts”	Provides an overview of the deployment process including deploying applications and libraries, configuring applications and libraries, and working with connected environments.
Chapter 8, “Deploying iPlanet UDS Applications and Libraries”	Describes the various ways you can configure and deploy applications and libraries. Explains how you package applications and libraries by making a distributions, and how you install applications and libraries in your environment.
Chapter 9, “Class Runtime Properties”	Explains how you set class runtime properties, which determine how objects of each class are treated at runtime.
Chapter 10, “Using Interfaces”	Provides complete information about how to use iPlanet UDS interfaces. Explains how you use interfaces with dynamic class loading and how you use interfaces for multiple interface inheritance.
Chapter 11, “Working With Service Objects”	Explains how you work with service objects including how you set the service object’s visibility and dialog duration, how you provide failover and load balancing, how you use reference partitions to share service objects with other applications, and how you set the service object’s environment search path.
Chapter 12, “Advanced Options for Structuring Client Applications”	Explains how you can design iPlanet UDS applications that use applets and nomadic clients.

Chapter	Description
Chapter 13, “Upgrading Deployed Applications”	<p>Describes how to upgrade iPlanet UDS user applications that are currently deployed. It covers advanced topics of interest to iPlanet UDS system managers as well as application developers, including:</p> <ul style="list-style-type: none"> <li>• when to use interoperable upgrades, compatibility level upgrades, and class version upgrades;</li> <li>• how to perform each type of upgrade;</li> <li>• how to use compatibility levels and class versions;</li> <li>• how to write converters to modify a deployed application;</li> <li>• and how to track changes in releases of an application.</li> </ul>
Chapter 14, “TOOL Reflection Classes”	<p>Introduces the TOOL classes that implement reflection. A running program can use reflection to examine or to change objects that are running in a local or remote partition.</p>
Chapter 15, “XSLT Processor Library”	<p>Provides a conceptual overview of the XSLT Processor library. The iPlanet Integration Server (iIS) Backbone product provides a built-in processor for performing such transformations between backbone-integrated applications. If you are not using iIS, you can use the TOOL XSLT Processor library to transform data exchanged between any TOOL applications.</p>
Chapter 16, “Source Code Management for iPlanet UDS Projects”	<p>Introduces the GenericRepository Library and the SourceCodeManager Library.</p>
Chapter 17, “Performance-Based Load Balancing”	<p>Explains how to implement performance-based load balancing.</p>
Chapter 18, “Creating HTTP Applications”	<p>Provides an overview of the HTTPSupport library.</p>
Chapter 19, “Enabling Security”	<p>Introduces the classes that implement Secure Sockets Layer (SSL) services. SSL is a standard security protocol that uses encryption and authentication techniques to protect communication on corporate intranets and internets. iPlanet UDS’s runtime services support the SSL connection as well as the building of certificates for encryption and authentication.</p>



Chapter	Description
Chapter 20, “Using the XMLDOM2 Library”	Describes the XMLDOM2 library, an API used to implement tree-based parsing of well-formed XML documents, enabling the programmatic manipulation of individual document elements.
Chapter 21, “Using the XMLSAX2 Library”	Describes the XMLSAX2 library, an API used to implement event-based parsing of well-formed XML documents, and generation of parse events for which application event handlers can register.
Chapter 22, “Accessing Internet Directory Services”	Describes the iPlanet UDS LDAP library, an API used to create, populate, modify, and query Internet directory services implemented with the Lightweight Directory Access Protocol.
Appendix A, “iPlanet UDS Example Applications”	Provides instructions on how to install the examples, a brief overview of the applications to help you locate relevant examples, and a section describing each example in detail.

## Text Conventions

This section provides information about the conventions used in this document.

Format	Description
<i>italics</i>	Italicized text is used to designate a document title, for emphasis, or for a word or phrase being introduced.
monospace	Monospace text represents example code, commands that you enter on the command line, directory, file, or path names, error message text, class names, method names (including all elements in the signature), package names, reserved words, and URLs.
ALL CAPS	Text in all capitals represents environment variables (FORTE_ROOT) or acronyms (UDS, JSP, iMQ).  Uppercase text can also represent a constant. Type uppercase text exactly as shown.

Format	Description
Key+Key	Simultaneous keystrokes are joined with a plus sign: Ctrl+A means press both keys simultaneously.
Key-Key	Consecutive keystrokes are joined with a hyphen: Esc-S means press the Esc key, release it, then press the S key.

## Other Documentation Resources

In addition to this guide, iPlanet UDS provides additional documentation resources, which are listed in the following sections. The documentation for all iPlanet UDS products (including Express, WebEnterprise, and WebEnterprise Designer) can be found on the iPlanet UDS Documentation CD. Be sure to read [“Viewing and Searching PDF Files” on page 44](#) to learn how to view and search the documentation on the iPlanet UDS Documentation CD.

iPlanet UDS documentation can also be found online at <http://docs.iplanet.com/docs/manuals/uds.html>.

The titles of the iPlanet UDS documentation are listed in the following sections.

## iPlanet UDS Documentation

- *A Guide to the iPlanet UDS Workshops*
- *Accessing Databases*
- *Building International Applications*
- *Escript and System Agent Reference Manual*
- *Fscript Reference Manual*
- *Getting Started With iPlanet UDS*
- *Integrating with External Systems*
- *iPlanet UDS Java Interoperability Guide*
- *iPlanet UDS Programming Guide*
- *iPlanet UDS System Installation Guide*
- *iPlanet UDS System Management Guide*
- *Programming with System Agents*
- *TOOL Reference Manual*
- *Using iPlanet UDS for OS/390*

## Express Documentation

- *A Guide to Express*
- *Customizing Express Applications*
- *Express Installation Guide*

## WebEnterprise and WebEnterprise Designer Documentation

- *A Guide to WebEnterprise*
- *Customizing WebEnterprise Designer Applications*
- *Getting Started with WebEnterprise Designer*
- *WebEnterprise Installation Guide*

## Online Help

When you are using an iPlanet UDS development application, press the F1 key or use the Help menu to display online help. The help files are also available at the following location in your iPlanet UDS distribution:

`FORTE_ROOT/userapp/forte/cln/*.hlp`.

When you are using a script utility, such as Fscript or Escript, type help from the script shell for a description of all commands, or help *<command>* for help on a specific command.

## iPlanet UDS Example Programs

A set of example programs is shipped with the iPlanet UDS product. The examples are located in subdirectories under `$FORTE_ROOT/install/examples`. The files containing the examples have a `.pex` suffix. You can search for TOOL commands or anything of special interest with operating system commands. The `.pex` files are text files, so it is safe to edit them, though you should only change private copies of the files.

## Viewing and Searching PDF Files

You can view and search iPlanet UDS documentation PDF files directly from the documentation CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

---

**NOTE** You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from <http://www.adobe.com>. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

---

► **To copy the documentation to a client or server**

1. Copy the `doc` directory and its contents from the CD-ROM to the client or server hard disk.

You can specify any convenient location for the `doc` directory; the location is not dependent on the iPlanet UDS distribution.

2. Set up a directory structure that keeps the `udsdoc.pdf` and the `uds` directory in the same relative location.

The directory structure must be preserved to use the Acrobat search feature.

---

**NOTE** To uninstall the documentation, delete the `doc` directory.

---

► **To view and search the documentation**

1. Open the file `udsdoc.pdf`, located in the `doc` directory.
2. Click the Search button at the bottom of the page or select Edit > Search > Query.
3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click Search.

A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

---

**NOTE** For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

---

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

All occurrences of the word or phrase on a page are highlighted.

5. Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

<b>Toolbar Button</b>	<b>Keyboard Command</b>
Next Highlight	Ctrl+] ]
Previous Highlight	Ctrl+[ [
Next Document	Ctrl+Shift+] ]

To return to the `udsdoc.pdf` file, click the Homepage bookmark at the top of the bookmarks list.

6. To revisit the query results, click the Results button at the bottom of the `udsdoc.pdf` home page or select Edit > Search > Results.

# How to Structure a Graphical User Interface

The graphical user interface of an iPlanet UDS application consists of windows. The basic procedures for creating windows are described in *A Guide to the iPlanet UDS Workshops*. This chapter describes what you need to consider when designing your windows, how to display them, and how to specify what happens when the end user interacts with them.

In this chapter, you will learn about the different types of windows, and how to:

- open and close windows
- display nested windows
- use dialog boxes
- write window event loops

For information on how to create tab folders, outline fields, list view fields, and tree view fields, see [Chapter 2, “Using Complex Widgets.”](#) For information on making your graphical user interface portable, see [Chapter 3, “Creating a Portable User Interface.”](#)

## About iPlanet UDS Windows

You use windows in iPlanet UDS to create a graphical user interface for an application or as page templates for printing. Before you create a window in the Window Workshop, you must consider how the window is going to be used by your application, because this will influence how you design the window, for example, whether you include a menu bar on the window, whether you create a generic window that can be reused, and so on.

The window you create in the Window Workshop can be any of the following:

**Standard window** A standard window is a new, independent window that you create for your user interface. To create a standard window, simply create a window class that is a subclass of the iPlanet UDS `UserWindow` class. This window class provides a empty window for which you create the form and menu bar from scratch.

**Inherited window** An inherited window is a window that inherits part of its appearance and behavior from an existing window. If you want your new window to inherit from an existing window, you can create a new window class that is a subclass of another custom window class (rather than the `UserWindow` class). When you define a subclass of a custom window class, your new window class inherits the form and menu bar from its superclass in addition to its methods, attributes, event handlers, and events. You can then extend the inherited form and menu bar by adding new widgets to the subclass window.

**Nested window** A nested window is a window that is designed to be displayed as part of one or more other windows. When you nest one window in another window, iPlanet UDS displays the nested window inside the main window so that it appears to be part of the main window. Creating a nested window is a good way to reuse a standard form or “subwindow” that is needed by several other windows.

**Page templates** A page template is a window that is designed specifically for printing. In the Window Workshop, creating a page template is the same as creating a window for a user interface—the fact that you are going to print the window instead of display it affects only the way you design the windows. The difference between printing a window and displaying a window is the way you handle the window in your code; you use the iPlanet UDS printing classes to print the window rather than opening it for display.

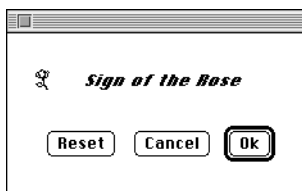
The following sections provide information about inherited windows, nested windows, and page templates.



## Inherited Windows

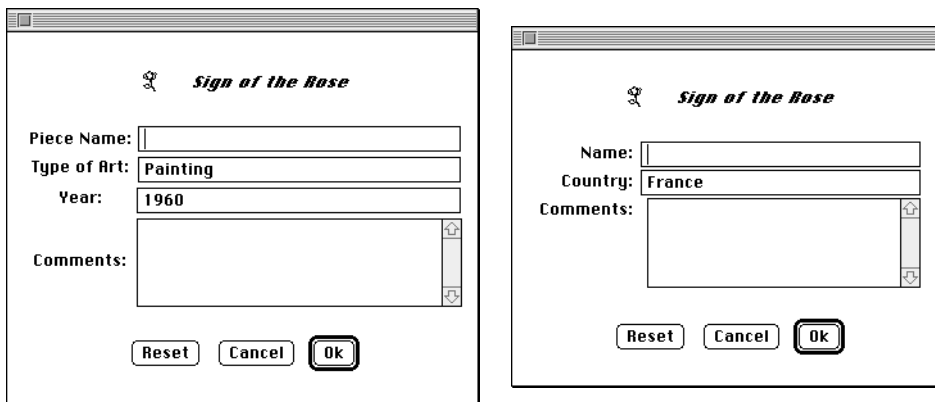
When you build a new window, you can create a *subclass window* that inherits its initial appearance and behavior from another window. The ability to create a subclass window enables you to define one window superclass that provides application or company-wide window formatting. You can then create subclasses of the generic window as many times as you wish, and customize each subclass window as appropriate for the application. The following example shows a simple data entry window skeleton with a logo and buttons.

**Figure 1-1** Example Superclass Window



The following example shows two subclass windows that have inherited the superclass window's widgets, and that also have additional data fields:

**Figure 1-2** Example Subclass Windows



To see these windows in context, see the iPlanet UDS example program `InheritedWindow`.

To create an inherited window, you declare your window class as a subclass of any existing window class. The new subclass window automatically inherits the form and menu bar defined for the existing window class (as well as the methods, attributes, events, and event handlers in the superclass). When you open the Window Workshop, the inherited window will be displayed. You can use the Window Workshop to add widgets to the inherited window or even to modify the inherited widgets.

iPlanet UDS also allows you to modify the superclass window at any time, even after creating the subclass windows. Any changes you make to the superclass are automatically inherited by its subclasses—you will see the inherited changes the next time you open the subclass window. If any of the subclasses have modified the inherited widgets, iPlanet UDS automatically merges the superclass changes with the subclass changes. Of course, you need to carefully coordinate changes made to the superclass window to ensure that the subclass windows still work properly. See the chapter on working with windows in *A Guide to the iPlanet UDS Workshops* for information about creating and modifying inherited windows.

## Named Event Handlers and Inherited Windows

iPlanet UDS named event handlers provide a very convenient way for you to provide event handling code in the superclass window that is inherited by the subclass windows. This works very effectively to allow both the widget and the widget's behavior to be defined in the superclass and then inherited by the subclasses. See *A Guide to the iPlanet UDS Workshops* for information about named event handlers and the Event Handler Workshop.

The following example shows the `ExitHandler` event handler for the `DataEntryWindow`, which is inherited by the `ArtistDataEntryWindow` and the `ArtDataEntryWindow`. This code provides the functionality associated with exiting the window.

```

-- Exit without validation. The Finalize Input box is not
-- checked in this PushButton's property sheet.
when <cancel_button>.Click do
  exit;

-- Validate the data. If ok, get out. The Finalize Input box
-- is checked in this PushButton's property sheet, forcing both
-- field and cross-field validation.
when <ok_button>.Click do
  exit;

-- No validation will occur on task shutdown.
when task.Shutdown do
  exit;

```

**Project:** InheritedWindow • **Class:** DataEntryWindow • **Event Handler:** ExitHandler

## Nested Windows

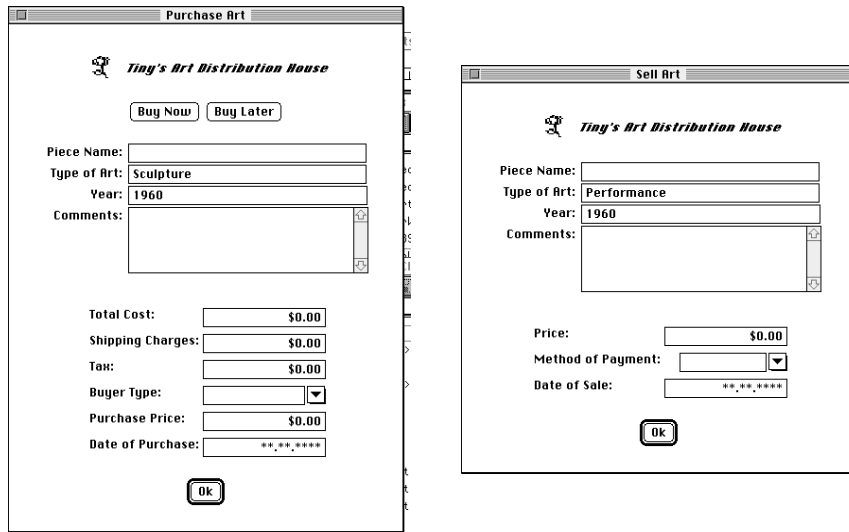
A nested window is a window that is designed to be displayed inside another window. When the application is running, the nested window appears to be a compound field on the main window. You can nest the same window on any number of other windows throughout your application (or in multiple applications). Nesting windows provides a way to reuse a standard “subwindow” (and the event handling code associated with it) in more than one window.

Figure 1-3 shows the original window as it was designed in the Window Workshop.

**Figure 1-3** Example Subwindow

Figure 1-4 shows how the nested window looks when the application is running and the nested window is displayed on top of two different main windows.

**Figure 1-4** Main Windows Displaying Nested Windows



To see these windows in context, see the iPlanet UDS example program `NestedWindow`.

You can make any window into a nested window. The only restriction is that the nested window cannot have a menu bar. Typically, you display the nested window within a compound field in the outer window, for example, within an empty cell of a grid field. When you are designing the nested window, you should consider exactly where it will be displayed on the outer window. You should take this into account when planning the size and content of the outer window.

After creating the nested window in the Window Workshop, you can display it on any number of other windows at runtime. To nest a window in another window, you set the value of the nested window's `Parent` attribute to the compound field in the outer window where you wish to display the nested window. For example:

**Code Example 1-1** How to Nest a Window

```

-- This fragment shows how to nest a window.
artObjectWindow : ArtObjectWindow = new;

-- Nest the Art Object Window. First assign a row
-- and column position. Give it a compound field as
-- a parent. Remove its frame so it will look like a
-- seamless part of the parent window.

artObjectWindow.Window.Row = 3;
artObjectWindow.Window.Column = 1;
artObjectWindow.Window.Parent = <main_grid>;
artObjectWindow.Window.FrameWeight = W_NONE;

self.Open();
...

```

**Project:** NestedWindow • **Class:** PurchaseWindow • **Method:** Display

Whenever you specify a parent for a window, iPlanet UDS displays the nested window without its frame (the title bar, close box, and so on) so that it does not look like a separate window. For information on how to write the Display method for a nested window, see [“Displaying a Nested Window” on page 61](#).

## Named Event Handlers and Nested Windows

iPlanet UDS named event handlers provide a very convenient way for you to define event handling code for the nested window that can be used by the window that displays the nested window. When the nested window definition includes named event handlers, the outer window can easily include the event handlers defined for the nested window as part of its own event handling code, making the two windows work together as one seamless window. See *A Guide to the iPlanet UDS Workshops* for information about named event handlers and the Event Handler Workshop.

Note that a dialog box is not the same thing as a nested window. A dialog box is simply a standard window without a menu bar. The way your program opens the dialog box, the format of the dialog box (usually containing OK and Cancel buttons), and the behavior of the dialog box (it usually blocks the window that opens it) is what differentiates a dialog from other windows in your user interface.

## Inherited Windows or Nested Windows?

Obviously, inherited windows duplicate some of the functionality of nested windows, that is, the ability to reuse window formatting and event handling code for more than one window. When should you use inherited windows? Inherited windows are most useful when you want to modify the form and menu inherited from the superclass. When should you use nested windows? Nested windows are most useful when you have a self-contained, modular form that you wish to reuse without modifications.

One advantage of inherited windows is the ability to view and customize the subclass windows in the Window Workshop. You cannot display a nested window on top of another window in the Window Workshop. Nested windows are displayed on their parent windows only when the application is running. To customize a single instance of a nested window, you must do so dynamically from your TOOL code.

## Windows as Page Templates

You can design a window to be used as a standard page in a report. The window functions as a form, which will be filled in with data at runtime. In your application, you can write TOOL code to fill in the data and print the report page. You can also prompt the end user to select printing options that control how the page is printed.

**Figure 1-5** illustrates a window in the Window Workshop that is intended for printing a report with header and footer information.

**Figure 1-5** Page Template Window



To see this window in context, see the ReportTemplateWindow class in the iPlanet UDS example program PrintSample.

A page template window can be any window class, either a standard subclass of `UserWindow` or any inherited window. There is nothing special you need to do when you create the page template window in the Window Workshop, except consider how the window layout will look when it is printed.

## Page Formatting

The window you design for printing can be any size. The `PrintPage` class allows you to “tile” a large window onto multiple pieces of paper. The `PrintPage` class provides attributes that control the page format, as well as a set of methods that allow you to draw text and graphics on the working page (see `PrintPage` class in the Display Library online Help for information.) Also, you can include any widgets on a page template, even widgets, such as array fields, that do not display all their data at once. The printing classes provide the option of expanding these fields at print time.

---

**NOTE** When printing a window with a menu bar, iPlanet UDS ignores the menu bar and prints the window without it. Therefore, if you are designing a window specifically for printing, you should not include a menu bar on it.

---

After you have created your page template in the Window Workshop, you can print it by using the printing classes in the Display library. See the `PrintDocument` class in the Display Library online Help for information about printing. [Figure 1-6](#) illustrates how the window shown above will look when it is printed. (Note that data repeats in the sample page because the sample data set is so small.)

While this section provides information about creating windows especially for printing, remember that you can allow your end user to print any iPlanet UDS window by providing a print command that uses the Display library printing classes to print the current window.

**Figure 1-6** Printed Version of Page Template

<b>Artist Report</b>		
Leonardo da Vin	Italy	His notebooks, unknown to his contemporaries, have revealed to the modern world his astonishing observations.
Henri Rousseau	France	He was naive, distressingly gullible, pompous and absurd.
Edgar Degas	France	His sculpture of the little 14-year old dancer was considered by Renoir to be the greatest sculpture of the century.
Jaspar Johns	United	His flags and targets were simple, but profound.
Pablo Picasso	Spain	His long and prolific life has left and indelible impression on the world of art.
Leonardo da Vin	Italy	His notebooks, unknown to his contemporaries, have revealed to the modern world his astonishing observations.
Henri Rousseau	France	He was naive, distressingly gullible, pompous and absurd.
Edgar Degas	France	His sculpture of the little 14-year old dancer was considered by Renoir to be the greatest sculpture of the century.
Jaspar Johns	United	His flags and targets were simple, but profound.
Pablo Picasso	Spain	His long and prolific life has left and indelible impression on the world of art.
Leonardo da Vin	Italy	His notebooks, unknown to his contemporaries, have revealed to the modern world his astonishing observations.
Henri Rousseau	France	He was naive, distressingly gullible, pompous and absurd.
Edgar Degas	France	His sculpture of the little 14-year old dancer was considered by Renoir to



# Structuring the User Interface

When you define your classes in the Project Workshop, you can create any number of windows. To display these windows, you must open them in your TOOL code. iPlanet UDS lets you open any number of concurrent windows. Usually, you display the application's first window (or windows) in the start method. After the first window is open, you can open more windows and close the open ones in response to the end user's commands. iPlanet UDS also provides prefabricated dialog boxes, such as a question dialog and a file selection dialog, that you can display over an open window.

Once a window is open, you must respond to the end user's interactions with it, such as entering data or selecting commands. The event loop that you write for the window provides code that is executed when the end user triggers events in the user interface.

The following sections provide information about opening and closing windows, using dialog boxes, and writing the window's event loop.

## Opening and Closing Windows

To display a window, you must invoke a method on a window object that opens the window.

### ► To display a window

1. Write a method for the window's class that opens the window.

Usually this method initializes the data for the window, opens the window on the screen, and handles the events from the window. Although this method does not need to be called "Display," iPlanet UDS provides a default Display method for your window classes that you can use for this purpose.

2. In the code where you plan to invoke your display method, create a window object.
3. When you are ready to display the window, invoke your display method, either directly or by using the **start task** statement to run a concurrent window.

The following sections provide information about these basic steps.

## Writing a Display Method

The New Window Class command in the Project Workshop provides a default Display method for the window. You can use this method to display the window or you can create your own display method with any appropriate name. Typically, the display method for a window initializes the data for the window and displays it on the end user's screen. In addition, it includes the **event loop** statement for the window, which handles all the events on the window.

### Code Example 1-2 A Typical Display Method

```
-- ... OverviewWindow Display method...
-- Actually open the window on screen
self.Open();
event loop
  when task.Shutdown do
    exit;
  when <CancelButton>.Click do
    exit;
end event;
-- Close the open window on screen.
self.Close();
```

### *Opening the Window*

After the data is initialized, you are ready to display the window to the end user. To do this, simply invoke the Open method of the UserWindow class. This displays the window on the end user's screen.

When the window is actually running, the **event loop** statement provides code that is executed in response to the end user's interactions with a window. This is where you respond to the end user's commands and actions. In the sample Display method, the **event loop** statement provides processing in response to the end user's click on the Cancel button. See ["Writing the Window Event Loop" on page 64](#) for information about writing this event loop.

### *Closing the Window*

After the event loop has completed, you can close the window by invoking the Close method on window object. The sample Display method uses this method to close the open screen. In addition, the AfterFinalize or Shutdown event is posted when the end user closes the window using the window system's close box. In your event loop, you can provide code that is executed in response to these events, so that, for example, the end user is prompted to save his changes before the window closes.

## Creating the Window Object

Before you can display a window to the end user, you must create the window object. You create a window object the same way you create any object, by using an *object constructor*.

The object constructor creates a new object of the specified class and optionally sets its initial values. At this point, you can set any of the window's attributes, such as the window usage, title, and so on. In the following example, `AddPaintingWindow` is a window class. The declaration for the `add_painting_window` variable constructs the window object.

### Code Example 1-3 Creating a Window Object

```
--from Display method of PaintingListWindow class
...
when <AuctioneerOnly.AddPaintingWindow>.Click do
  add_painting_window : AddPaintingWindow = new;
  add_painting_window.Window.Title =
    TextData(value = 'New painting');
...
```

## Displaying the Window

iPlanet UDS lets you display any number of windows at the same time. When a window is being displayed, it can be in one of two states:

State	Description
active	The user can interact with the window (for example, enter data or give commands).
inactive	The user cannot interact with the window. When the user moves a cursor on top of an inactive window, it changes to indicate that the window is currently inactive.

When a window is active and you want to open a second window, you have the following choices:

- Make the first window inactive. The new window *blocks* the first window. The end user can interact with the new window, but not with the first window.
- Leave the first window active. This creates two *concurrent windows*. The end user can interact with either window.

## Blocking the Calling Window

If you want the new window to block the first window, simply invoke the `Display` method on the new window. Because the invoking method waits while the invoked method is executing, the end user will only be able to interact with the new window. The calling window will be inactive (it accepts no input from the end user) until the `Display` method completes and control returns to the invoking method. The first window will then become active again.

The following example illustrates invoking the `Display` method to display a window that blocks the calling window.

### Code Example 1-4 Blocking the Calling Window

```
-- From Display method of PaintingListWindow class.  
...  
when <AuctioneerOnly.AddPaintingWindow>.Click do  
  add_painting_window : AddPaintingWindow = new;  
  add_painting_window.Window.Title =  
    TextData(value = 'New painting');  
  -- Synchronous invocation will block  
  add_painting_window.Display  
    (auctionMgr = self.AuctionMgr);  
...
```

Because the invoking method waits for the `Display` method to complete, you can use the method's return value and output parameters to communicate with the calling window.

## Creating Concurrent Windows

In iPlanet UDS, you create concurrent windows by using multitasking. If you invoke the Display method on a window using the start task statement, the first window and the second window will run as independent tasks. Because they are separate tasks, the end user can interact with both windows simultaneously.

### Code Example 1-5 Using the start task Statement to Display a Window

```
-- From Display method of PaintingListWindow class.
...
when <AuctioneerOnly.AddPaintingWindow>.Click do
  add_painting_window : AddPaintingWindow = new;
  add_painting_window.Window.Title =
    TextData(value = 'New painting');
  -- Asynchronously, both windows will be active
  start task add_painting_window.Display
    (auctionMgr = self.AuctionMgr);
...
```

When you run two windows as separate tasks, use events to communicate between the windows. For example, an update in one window can post an event. The second window can then respond to the event by making corresponding updates.

## Displaying a Nested Window

A special kind of concurrent window is a nested window. If you nest one window in another window, iPlanet UDS displays the nested window inside the parent window so that it becomes part of the parent window. Because the two windows are concurrent, the end user can interact with both windows as if they were one; in fact, the end user probably will not be able to tell they are separate windows. The nested window appears to be a compound field on the parent window. This provides a way to reuse a standard “subwindow” (and the event handling code associated with it) in more than one window.

To nest a window in another window, simply set the value of the nested window’s Parent attribute to a field on the window where you wish to display the nested window. You can make any existing window into a nested window. The only restriction is that the nested window cannot have a menu bar.

The Display method for the outer window should do the following:

1. Create the outer window object.
2. Create the nested window object.
3. Set the Parent attribute of the nested window object to be any compound field on the outer window.
4. The event loop for the outer window should use the **register** statement to register the named event handlers for the nested window.

Whenever you specify a parent for a window, iPlanet UDS displays the window without its frame (the title bar, close box, and so on) so that it does not look like a window.

Any window whose Parent attribute is set to NIL does not have a parent window and is therefore considered a *main window*. iPlanet UDS displays a main window with its frame so that it looks like a standard window. A main window can have a menu bar.

The following example illustrates the Display method for a window that displays a nested window:

**Code Example 1-6**    Displaying a Nested Window

```
begin
-- This method shows how to nest a window and register its
-- event handler.

artObjectWindow : ArtObjectWindow = new;

-- Nest the Art Object Window. First assign a row
-- and column position. Give it a compound field as
-- a parent. Remove its frame so it will look like a
-- seamless part of the parent window.

artObjectWindow.Window.Row = 2;
artObjectWindow.Window.Column = 1;
artObjectWindow.Window.Parent = <main_grid>;
artObjectWindow.Window.FrameWeight = W_NONE;

self.Open();

event loop

  preregister
    -- Include the ArtObjectWindow's event handler in this
    -- event loop.
    register artObjectWindow.artObjectHandler
      (artType = 'Performance');
```

**Code Example 1-6** Displaying a Nested Window (*Continued*)

```

    when task.Shutdown do
      exit;
    ...
  end event;
  self.Close();

```

**Project:** NestedWindow • **Class:** SellWindow • **Method:** Display

## Using Dialog Boxes

iPlanet UDS provides five dialog boxes that you can display over a window. These are useful for displaying error messages or warnings, prompting a user to answer a question, or enabling the end user to select a file. All five dialog boxes automatically block the rest of the windows in the application (that is, all the windows in the application are inactive until the dialog box is closed).

The dialog boxes are:

Dialog Box Type	Description
file selection	Displays a list of files from which the end user can make one selection.
message	Displays the message you specify along with an icon indicating the message type.
question	Displays the question you specify, along with buttons the end user can click to answer the question.
print	Displays the window system's standard print dialog.
print setup	Displays the window system's standard print (or page) setup dialog.

To display one of these dialog boxes, you must invoke the appropriate Window class method on the window object. The dialog box is centered over the window, and its size is adjusted to fit the contents.

In the following example, the Display method displays a message dialog, which explains an error condition to the user.

**Code Example 1-7** Using Window.MessageDialog

```
-- From Display method of PaintingListWindow class
...
when <BidderOnly.BecomeAuctioneerButton>.Click do
  case self.AuctionManager.RequestAuctioneerStatus
    (password=self.AuctioneerPassword,
     name=self.UserName) is
    when 1 do
      -- Bad password
      self.Window.MessageDialog
        ('Bad password.
         Enter new password and try again.');
```

See the Display Library online Help for details on these methods.

## Writing the Window Event Loop

To respond to the end user's interactions with a window, you use an event loop in a method on the window. For example, a **when** clause in your event loop can respond to the Click event on a Close button by prompting the end user to save his changes and then closing the window.

When you select the events that you wish to respond to, remember that one action by the end user typically triggers two or more events. First of all, there is the basic event on the field itself. Second, iPlanet UDS raises *child events* for all the parent fields.

Under certain circumstances, one end user action triggers a whole series of events, called an *event chain*. For example, when the end user moves the input focus from one field to another, this triggers a chain of events, which affects both the field that is being left as well as the field to which the end user is moving. These events are always triggered in a certain, fixed order. You can respond to any or all of them in your code.



## Using Event Handlers

Remember that besides listing events directly in the **event** statement, you can use the **register** statement to include a named event handler within the **event** statement. The event handler provides reusable, modular event handling code that you can include in any number of **event** statements. The discussions in this section about the events triggered by end user's interaction with the user interface are as relevant for named event handlers as for **event** statements.

The following sections provide further information on working with child events, along with details about the three event chains: the input focus event chain, the field selection event chain, and the close window event chain. This discussion is followed by information about writing the event loop for an inherited window and for a window that displays a nested window.

## Child Events

Every time an event is triggered on a field, iPlanet UDS raises a corresponding event for all the parents of that field. For example, when the end user clicks the push button contained by the panel, this triggers a Click event for the push button and a ChildClick event for the panel (meaning one of the panel's children was clicked). Because the window is the parent for all the widgets that it contains, this also posts a ChildClick event for window. The child events are always triggered in a certain, fixed order. The original event is always first, followed by the child event for the inner-most parent, moving "out" to the outermost parent (the window itself).

Menu widgets also produce child events. For example, when a menu command gets an Activate event, all its parent compound menus receive corresponding ChildActivate events.

For event chains (described below), iPlanet UDS triggers all the child events for the first event in the chain before moving on to the second event in the chain.

## Input Focus Event Chain

iPlanet UDS triggers a chain of events when the end user moves from a field that has the input focus to another field that can accept the input focus. The purpose of this event chain is to let you detect when the first field is being left and the target field is being entered. This is most useful for doing data validation on a set of related fields within a compound field. Since you can detect when the end user leaves any field in the compound field, you can do data validation for the entire compound field each time the user makes a change.

A field that can accept the input focus is a field that can accept the input from the keyboard. Your window system indicates which field has the input focus by displaying the text insertion cursor or by using another indicator. Only certain fields can have the input focus, and this differs between window systems. For example, all window systems give the input focus to text fields. However, only some window systems give the input focus to push buttons. We therefore recommend that you use the `AfterValueChange` event for data validation rather than the `AfterFocusLoss` event.

The chain of events that is triggered when the input focus changes fields affects both the field that is being left and the target field. The purpose of the event chain is to allow you to validate the data in the field the end user is leaving and, if necessary, prevent the end user from leaving the field. Event chains are useful mainly for the character fields (text field, text edit field, data field, and fillin field).

When the end user moves the input focus from one field to another, the following events are posted (in order):

1. `AfterValueChange` (for the field being left).

This event is triggered only if the end user actually changed the data. At this point, the focus is still in the field the user is leaving.

2. `ChildAfterValueChange` (for all parents of the field being left).

At this point, the focus is still in the field the user is leaving.

3. `BeforeFocusLoss` (for field being left).

At this point, the focus is still in the field the user is leaving.

4. `BeforeFocusLoss` (for any parent of the field being left that does not also contain target field).

At this point, the focus is still in the field the user is leaving.

5. `AfterFocusGain` for target field.

The focus is now in the target field.

6. `AfterFocusGain` (for any parent of the target field that does not also contain the field being left).

The focus is now in the target field.

7. Traverse event (for all compound fields that contain both the previous field and the target field).

This event indicates there has been a change of focus within the compound field. The focus is now in the target field.

If the target field cannot accept the input focus, the focus events and Traverse events are *not* triggered. Because only certain fields can have the input focus, and this differs between window systems, we recommend that you use the AfterValueChange event (rather than the BeforeFocusLoss event) to do the data validation for all fields except the character fields.

If the data is not valid and you want to prevent the focus from moving to the target field, you can use the PurgeEvents method on the window to remove the remaining events in the chain from the event queue. If you invoke this method in response to the BeforeFocusLoss event, this removes the AfterFocusGain and Traverse events from the queue and the focus remains in the original field.

The Traverse event is very useful for performing data validation for a section of the window. However, because there is no guarantee that the end user will interact with the fields in sequential order, you must still do data validation for the entire window before you save changes or close the window (see [“Close Window Event Chain” on page 68](#) for information on this).

## Field Selection Event Chain

Another important event chain occurs whenever the end user selects a field that is in a selectable state. This effects both the field that was previously selected (and is now “unselected”) as well as the field that the end user is selecting. This event chain is very useful for keeping track of which fields on the window are currently selected. For example, you may wish to activate or deactivate menu items based on the types of fields that are currently selected.

When the end user selects a field or group of fields, the following events are posted (in order):

1. AfterDeselect event for the all fields that are no longer selected.
2. ChildAfterDeselect events for the parents of all fields that are no longer selected.
3. AfterSelect event for the field or fields being selected.
4. ChildAfterSelect event for all the parents of the field or fields being selected.
5. SelectionChanged event for the window. This event indicates that the window’s selection list (stored in the SelectedFields attribute) has changed.

Note that if the field is not already selected, moving or resizing a field will automatically select the field and therefore trigger the field selection event chain.

## Close Window Event Chain

A special event chain is triggered when the user closes the window using the window system's close box. The purpose of this chain is to ensure that data validation is done for the last field that the user modified before the window is actually closed. For information about this, see the `RequestFinalize` method on the `Window` class in the Display Library online Help.

## Event Loop for Inherited Windows

When you write the event loop for an inherited window, you need to provide event handling code for the widgets that the window has inherited from its superclass. Defining named event handlers in the superclass window provides a very easy solution to this problem.

When the superclass window provides named event handlers for the events on its widgets, the subclass windows can easily inherit all the event handling code they need for the inherited widgets. To use the inherited event handlers, a subclass window uses the **register** statement within its event loop. The subclass windows that inherit the `ExitHandler` event handler can add this event handling code to their event loops by using the **register** statement.

The following example illustrates an event loop for an inherited window that uses the **register** statement to register named event handlers from its superclass:

### Code Example 1-8 Registering Named Event Handlers for a Superclass

```
-- This window is inherited from DataEntryWindow. The fields
-- related to the ArtObject object were added. This event loop
-- registers event handlers on self (that relate to the
-- art object) and event handlers on super (which handle
-- generic button events).
event loop
  preregister
    register self.ArtObjectHandler();
    register self.ResetHandler();
    -- Use the inherited event handler to handle exit
    -- button events.
    register super.ExitHandler();
end event;
```

**Project:** InheritedWindow • **Class:** ArtDataEntryWindow • **Method:** Display

## Event Loop Using a Nested Window

When you write the event loop for a window that is displaying a nested window, you need to provide event handling code for the widgets on the nested window. Defining named event handlers in the nested window is a good way to provide event handling code that can be reused by any other window that runs the nested window.

When the nested window provides event handlers for the events on its widgets, you can simply include these event handlers within the event loop for the outer window by using the **register** statement. For example, if the nested window defines an `artObjectHandler` event handler, you can include the `artObjectHandler` event handler within the outer window's event loop as follows:

### Code Example 1-9 Using the register Statement

```
event loop

  preregister
    -- Include the ArtObjectWindow's event handler in this
    -- event loop.
    register artObjectWindow.artObjectHandler
      (artType = 'Performance');

  when task.Shutdown do
    exit;
```

**Project:** NestedWindow • **Class:** SellWindow • **Method:** Display



# Using Complex Widgets

This chapter describes how to use some of iPlanet UDS's complex widgets, including detailed information about creating the widgets in the Window Workshop, assigning data to the widgets, and manipulating the widgets dynamically in your TOOL code.

The following widgets are covered:

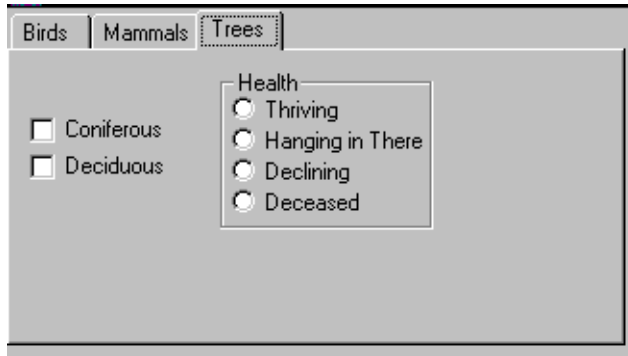
- tab folders
- outline fields
- list view fields
- tree view fields

## About Tab Folders

A tab folder is a widget that displays one or more pages with labeled tabs. The end user views one tabbed page at a time by clicking on the tab for the page he wishes to display. Typically, a tab folder provides a set of dialogs, each of which displays a separate group of properties or settings.

Figure 2-1 illustrates a tab folder:

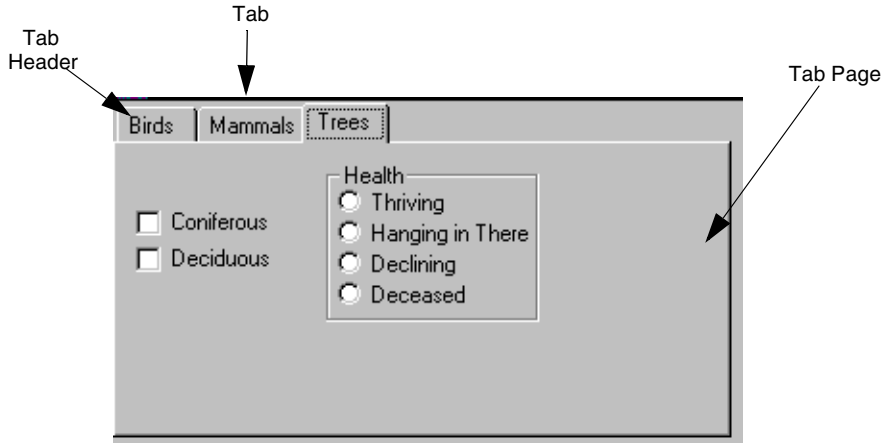
**Figure 2-1** Tab Folder



In iPlanet UDS, a tab folder is a compound widget that consists of an array of panels. Each panel has a caption associated with it, and the panel's caption is used as the tab label. The contents of the panel provide the contents of the tab page. When a panel is included in a tab folder widget, it is referred to as a "tab page." However, in the Window Workshop you can open the panel's properties dialog while it is in the tab folder and set any of the panel properties as usual.

Figure 2-2 illustrates the tab folder components:

**Figure 2-2** Tab Folder Components





The end user starts interacting with a tab folder by clicking on the tab for the page he wishes to display. After the selected page is displayed, the end user can then use the mouse to move freely from one individual field on the page to another. On Windows, iPlanet UDS provides mouseless support for tab folders—the end user can use the standard keys that Windows provides for the native tab folder control to navigate through the tab page.

The following sections provide detailed information about creating tab folders in the Window Workshop and creating tab folders dynamically. The Display Library online Help provides complete information on the TabFolder class.

## Creating Tab Folders in the Window Workshop

There are two ways to create a tab folder in the Window Workshop:

- Use the Widget > New > TabFolder command or New Tab Folder tool.

When you use the Widget > New > TabFolder command or the Tab Folder tool on the palette, iPlanet UDS creates a default tab folder, with three empty tab pages. You can then design each of the tab pages, and add or remove tab pages if desired.

- Use the Widget > Group Into > Tab Folder command.

Before using the Widget > Group Into > Tab Folder command, you must create the panels or other widgets you wish to include in the tab folder. After grouping the widgets into a tab folder, you can edit them as you wish.

The following two sections provide detailed instructions for using these two commands.

### Using the New > TabFolder Command

The New > TabFolder command creates a tab folder widget that contains three empty tab pages. The tabs have the following default labels: Tab 1, Tab 2, and Tab 3.

After creating the tab folder, you can design each of the tab pages by adding widgets to them as you would to any panel. The Window Workshop allows you to select the tab page you wish to edit by clicking on its tab—clicking on its tab brings the tab page to the top.

You can also add or delete tab pages, and reorder the pages as you wish. For information about adding, deleting, and reordering tab pages, see [“Editing the Tab Folder” on page 76](#). For information on setting the properties on the tab folder itself, see [“Setting Tab Folder Properties” on page 79](#).

► **To create a new tab folder**

1. Choose the Widget > New > TabFolder command or click the New Tab Folder tool.
2. On the form, draw a rectangle to indicate the tab folder's size.
3. Design each of the tab pages by adding widgets to them. Use a grid field within the panel to maintain portability.

### *Adding Tab Labels*

The label on an individual tab is determined by the Caption property for the panel on the tab page. When you create a new tab folder, iPlanet UDS provides default labels for each tab. To reset the default labels, you must use the Caption property for the individual panels. See [“Editing the Tab Folder” on page 76](#) for information.

By default, the tabs for the tab folder are set with the following properties:

- scrolling  
If there are more tabs than can fit across the tab folder, iPlanet UDS provides a horizontal scroll mechanism. On Windows NT 4.0/3.51, you can reset this property to provide multirow tabs.
- packed  
The Packed setting creates tabs that are sized individually to accommodate their labels. You can reset this property to provide evenly spaced tabs. The Evenly Spaced setting creates tabs with a uniform size—the tab size is determined by the longest label.

For information about setting the tab folder properties, see [“Setting Tab Folder Properties” on page 79](#).

### **Using the Group Into > TabFolder Command**

The Group Into > TabFolder command lets you group existing panels, compound widgets, and simple widgets into a tab folder. The Group Into > TabFolder command creates tab pages from the selected widgets as follows:

- every panel becomes an individual tab page
- every compound widget is grouped into a new panel, which in turn, becomes an individual tab page
- all simple widgets are grouped into a single panel, which in turn, becomes a single tab page

The order of the tab pages in the tab folder is determined by the order in which the widgets were originally added to the form. The first compound widget is the first tab page, the second compound widget is the second tab page, and so on. After creating the tab folder, you can reorder the tab pages as you wish (see [“Editing the Tab Folder” on page 76](#) for information).

Generally, you design the tab pages in the form of panels before giving the Group Into > TabFolder command. Be sure to use grid fields within the panels to ensure portability. If you wish to specify the tab label for each panel before giving the Group Into command, you can do so by setting each panel’s Caption property on the panel’s properties dialog.

► **To create a tab folder**

1. On your form, create the widgets you wish to use as or include on tab folder pages.
2. Select all the widgets you want to include in the tab folder.
3. Choose the Widget > Group Into > TabFolder command.

After creating the tab folder, you can edit each of the tab pages by adding widgets, deleting widgets, or changing widgets on them as you would on any panel. The Window Workshop allows you to select the tab page you wish to edit by clicking on its tab—clicking on its tab brings the tab page to the top.

You can also add or delete tab pages, and reorder the pages as you wish. For information about adding, deleting, and reordering tab pages, see [“Editing the Tab Folder” on page 76](#). For information on setting the properties on the tab folder itself, see [“Setting Tab Folder Properties” on page 79](#).

The Group Into > TabFolder command uses a default size for the tab folder. After creating the tab folder, you can resize it by using the widget’s resize handles or by using the Widget > Size Properties... command.

The label on an individual tab is determined by the Caption property for the panel on the tab page. When you group widgets into a tab folder, iPlanet UDS provides a default caption for each panel that does not already have one. To reset the default captions, you must use the Caption property for the individual panels. See [“Editing the Tab Folder” on page 76](#) below for information.

By default the tabs for the tab folder are set with the following properties:

- scrolling

If there are more tabs than can fit across the tab folder, iPlanet UDS provides a horizontal scroll mechanism. On Windows NT 4.0/3.51, you can reset this property to provide multirow tabs.

- packed

The Packed setting creates tabs that are sized individually to accommodate their labels. You can reset this property to provide evenly spaced tabs. The Evenly Spaced setting creates tabs with a uniform size—the tab size is determined by the longest label.

For information about setting the tab folder properties, “[Setting Tab Folder Properties](#)” on page 79.

## Editing the Tab Folder

You can use the Edit menu to copy, cut, delete, and paste the individual tab pages within the tab folder the same way you use the Edit menu commands to manipulate other widgets.

After creating a tab folder, you can make any of the following edits:

- add new tab pages
- delete tab pages
- reorder the pages in the tab folder
- change the labels on the tabs

### *Adding a Tab Page*

There are two different ways you can add a new tab page to a tab folder: copy an existing tab folder page or create a new panel.

Copying an existing tab page is useful when the basic design of your tab pages is similar. When you copy an existing tab page, everything on the tab page is duplicated, including the tab label and all widgets on the panel. To copy an existing tab page, you can use the Edit > Copy command or the Edit > Duplicate command. Using the Edit > Copy command enables you to paste the new tab page exactly where you want it. The Edit > Duplicate command always inserts the new tab page at the beginning of the set.

► **To copy an existing tab page with the Edit > Copy command**

1. Select the tab page you wish to copy.
2. Choose the Edit > Copy command.
3. Choose the Edit > Paste command.
4. Indicate the position for the new tab page.

You can either select a tab page for the new page to precede or you can select a tab page for the new tab page to follow. To indicate which tab page you want your new tab page to precede, click on the left side of the tab. To indicate which tab page you want your new tab page to follow, click on the right side of the tab.

5. Edit the new tab page as desired.

Creating a new panel is useful when you want to design the tab page format from scratch. To provide the label for the new tab page, you can set the panel's Caption property.

► **To create a new tab page**

1. Place the simple and compound widgets for the tab page directly on the form.
2. Choose the Widget > Group Into > Panel command to group the widgets into a panel.
3. Choose the Edit > Cut command.
4. Choose the Edit > Paste command.
5. Indicate the position for the new tab page.

You can either select a tab page for the new page to precede or you can select a tab page for the new tab page to follow. To indicate which tab page you want your new tab page to precede, click on the left side of the tab. To indicate which tab page you want your new tab page to follow, click on the right side of the tab.

The Header Style property (described under [“Setting Tab Folder Properties” on page 79](#)) lets you control how the user views the tabs, with scrolling or in multiple rows. However, we strongly recommend that you limit the number of tabs so that all the tabs are visible without using scrolling or multiple rows.

### *Deleting a Tab Page*

To delete a tab page, you simply remove the panel from the tab folder using the Edit > Delete command.

#### ► **To delete a tab page**

1. Click the tab for the page you wish to delete.  
Clicking the tab brings the selected page to the top of the tab folder.
2. Select the tab page you wish to delete by selecting its panel.
3. Choose the Edit > Delete command.

If you wish to remove the page from the tab folder but save it for use elsewhere, you can use the Edit > Cut command, and then paste the panel wherever you want it.

### *Reordering Tab Pages*

To reorder the tab pages in a tab folder, you must use the Edit > Cut and Paste commands. Use the Cut command to remove the tab page you wish to move, and use the Paste command to paste the tab page into the correct position.

#### ► **To move a tab page**

1. Select the tab page you wish to move.
2. Choose the Edit > Cut command.
3. Choose the Edit > Paste command.
4. Indicate the new position for the tab page.

You can either select a tab page for the moved page to precede or you can select a tab page for the moved tab page to follow. To indicate which tab page you want your moved tab page to precede, click on the right side of the tab. To indicate which tab page you want your moved tab page to follow, click on the left side of the tab.

### Editing the tab Label

To edit a tab label, you must reset the Caption property for the top-most panel.

#### ► To edit a tab label

1. Double-click the panel, or select the panel and choose the **Widget > Properties...** command.
2. On the Panel Properties dialog, enter the tab label in the Caption field.

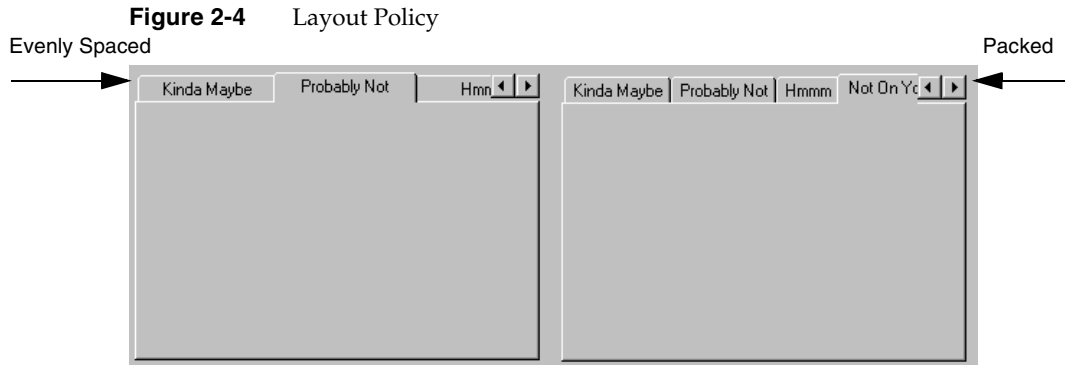
### Setting Tab Folder Properties

Tab folders have two special properties that let you specify how the tabs are sized and displayed: Header Style and Layout Policy.

By default, when there are more tabs than can fit on the tab header at one time, the tab folder provides a horizontal scroll bar to let the end user view all the tabs. On Windows NT 4.0/3.5.1, you have the option of displaying multirow tabs. The Multirow setting for the Header Style property displays the tabs in multiple rows, which lets the end user view all the tabs at once and provides the appearance of a file drawer. The Scrolling setting (the default) for the Header Style property provides a horizontal scroll bar which the end user can use to scroll through the tabs.



The Layout Policy property specifies whether the tabs are packed or evenly spaced. The Packed setting (the default) creates tabs that are sized individually to accommodate their labels. The Evenly Spaced setting creates tabs with a uniform size—the tab size is determined by the longest label.

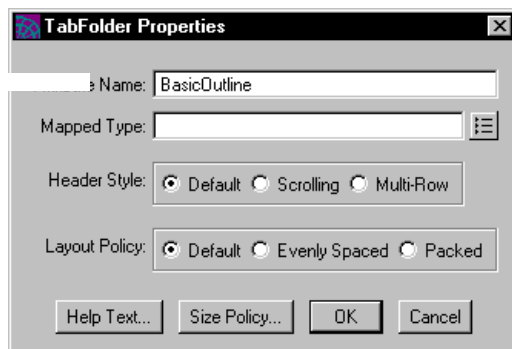


### Selecting the Tab Folder

To set the properties on the tab folder, you must select the tab folder widget. You can do this by clicking on the background of the tab folder (click in the area to the right of the last tab) or by using Ctrl-Click to select the parent of one of the tab pages (a panel).

The following figure illustrates the TabFolder Properties dialog:

**Figure 2-5** TabFolder Properties Dialog





The following table describes the tab folder properties.

<b>Use This Property</b>	<b>For This Purpose</b>
Attribute Name	To set an attribute name for the tab folder.
Mapped Type	To set the tab folder to map to a class. This is an optional setting; you need not map a tab folder to a class. However, if you do map the tab folder to a class, the named attributes of the widgets contained by the tab folder must match the attribute names and types of the tab folder's class.
Header Style	To specify how the tabs are displayed when there are more tabs than can fit on one line across the top of the widget, Scrolling or Multirow.
Layout Policy	To specify how the size of the tabs is determined, Evenly Spaced or Packed.
Help Text	To open the Help Text dialog.
Size Policy	To open the Size Policy dialog.

## Creating Tab Folders Dynamically

To create a tab folder dynamically, you use the `SetPages` method of the `TabFolder` class. The `SetPages` method adds an array of panels to the tab folder. Each panel in the array becomes an individual tab page, the `Caption` attribute for each panel provides the label for each tab, and the order of the panels within the array determines the order of the tab pages within the tab folder. See the [Display Library online Help](#) for information on the `SetPages` method.

### ► To create a tab folder

1. Create a `TabFolder` object.
2. Create an array of panels.
3. For each panel in the array, add the widgets you wish to display on the tab page.

Be sure to use grid fields to align the widgets within the panel—this ensures portability.

4. For each panel in the array, set the Caption attribute to the label you wish to display on the tab.
5. Use the SetPages method to add the array of panels to the tab folder.
6. Load the data into the tab folder, either into all the pages at once or into the first page only. (See below for further information on this.)

The following code sample illustrates creating a tab folder dynamically:

**Code Example 2-1**    Creating a Tab Folder Dynamically

```
-- Create a TabFolder dynamically.
foTabFolder : TabFolder = new;
panelArray : array of Panel = new;
index : integer;

-- These three panels were created in the Window Workshop.
panelArray[1] = <BirdPanel>;
panelArray[2] = <MammalPanel>;
panelArray[3] = <TreePanel>;

-- SetPages moves the panels into the TabFolder. They will appear
-- on the window only within the TabFolder.
foTabFolder.SetPages(pages = panelArray);

-- Set the header style.
foTabFolder.HeaderStyle = TH_MULTIRow;

-- You must always parent the TabFolder.
tabFolder.Parent = self.Window.Form;

self.Open();
event loop
    ...
```

**Project:** TabFolders • **Class:** DynamicTabFolderWindow • **Method:** Display

### *Loading Data into the Tab Folder*

You can load the data into the tab folder using two different techniques. The simplest technique is, on window start-up, to load the data for all the tab pages at once. The end user can then freely move from one tab page to another as desired. This technique requires no special coding.

The second technique is, on window start-up, to load the data into the first tab page only. Then, as the end user clicks on a given tab, you can load the data into the corresponding tab page. This provides improved performance at start-up, although it requires extra coding. First, you must handle the `AfterTabSelect` event, loading the data into the tab page when the end user clicks on the corresponding tab. And second, you should set a flag so you do not load the data for a given tab page more than once.

The `AfterTabSelect` event is posted on the tab folder when the end user clicks on one of its tabs. By default, when the end user clicks on a tab, iPlanet UDS automatically moves the selected tab page to the top of the folder, where the end user can interact with it. If all the data is already loaded into the tab page and you do not wish to provide special processing, you do not need to handle the `AfterTabSelect` event. However, you might wish to provide special processing when the end user clicks on the tab, for example, to load data from a database or external system that you wish to display on the tab page only when the end user actually selects that tab page. In this case, you can explicitly handle the `AfterTabSelect` event (see the Display Library online Help for information):

### Code Example 2-2 Handling the `AfterTabSelect` Event

```
-- When the user selects the Mammal Tab Page, display a message.
when foTabFolder.AfterTabSelect(Page = Page) do
  if Page = <MammalPanel> then
    self.Window.MessageDialog(
      messageText = 'Congratulations on seeing a mammal.',
      messageType = MT_INFO);
  end if;
```

**Project:** TabFolders • **Class:** DynamicTabFolderWindow • **Method:** Display

### *Controlling Tab Size and Appearance*

To control the size and appearance of the tabs on the tab folder, you can use the `HeaderStyle` and `LayoutPolicy` attributes. The `HeaderStyle` attribute specifies whether the tabs provide a horizontal scroll bar (the default) or whether they are displayed in multiple rows. The `LayoutPolicy` attribute specifies whether the tabs are uniformly sized to fit the longest tab label or whether the tabs are individually sized to fit their labels (the default).

Although the `HeaderStyle` attribute lets you control how the user views the tabs when there are more tabs than can fit on one row, we strongly recommend that you limit the number of tabs so that all the tabs are visible without using scrolling or multiple rows.

### *Adding, Deleting, and Replacing Tab Pages*

After creating a tab folder, you can add, delete, and replace tab pages from your TOOL code. The `AddPage` method inserts a new tab page into the tab folder at the specified position. The `DeletePage` method removes the specified tab page from the tab folder. The `ReplacePage` method replaces the specified tab page with a new tab page.

### *Merging Two Tab Folders*

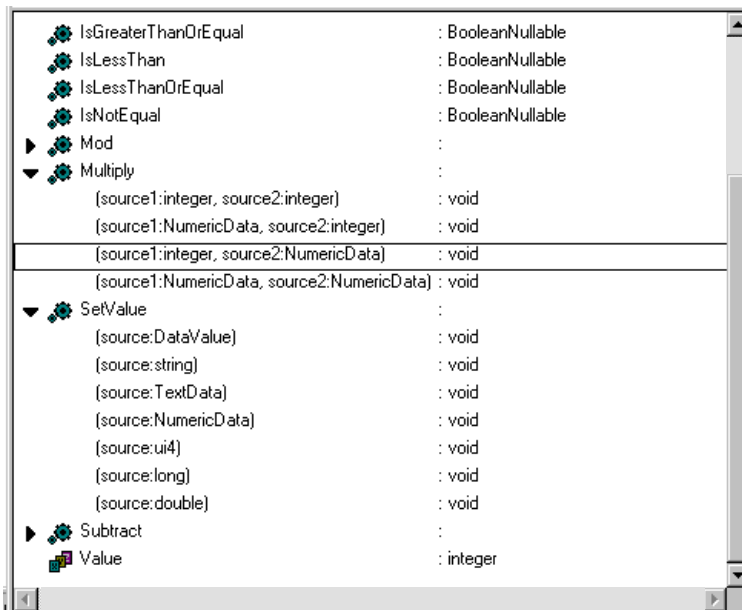
If you wish to merge two tab folders into one, you can use the `GetPages` method to return the arrays of panels in both of the existing tab folders. You can then merge both these arrays into a single new array. After preparing the new array as necessary (setting `Caption` attributes and so on), use the `SetPages` method to transfer the panels in the new array into a tab folder.

See the Display Library online Help for complete information on the attributes and methods available on the `TabFolder` class.

## About Outline, List View, and Tree View Fields

iPlanet UDS outline, list view, and tree view fields allow you to create the standard browsers your end users have grown accustomed to in a wide variety of window-based applications.

An *outline field* provides a browser for multi-column data. It can display a hierarchy of data as an indented outline, or it can display one level of multi-column information. If the data structure is larger than the outline field, the outline field provides a scrollbar that lets the end user scroll through the data. The following figure illustrates an outline field:

**Figure 2-6** Outline Field

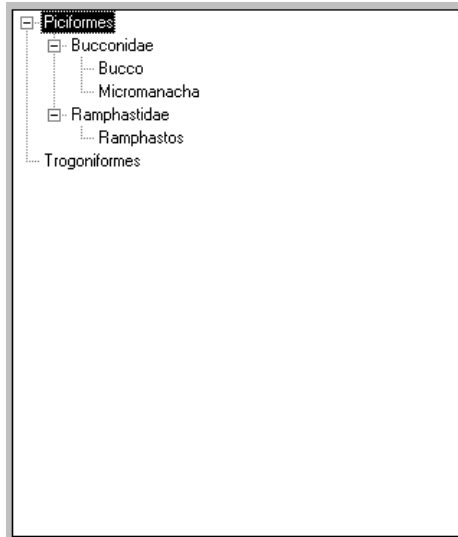
A *list view field* displays a set of items, each consisting of an icon with a label, from which the end user can make selections. There are four styles for list view fields: image (large icon), small icon, list, and detail. The following figure illustrates a detail style list view field:

**Figure 2-7** List View Field

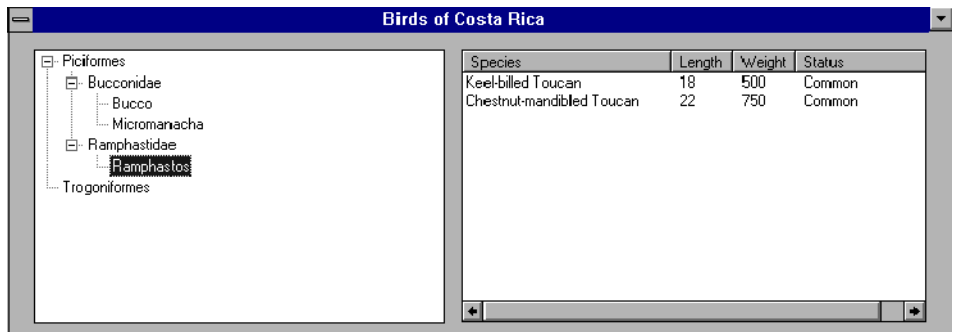
Species	Length	Weight	Status
Keel-billed Toucan	18	500	Common
Chestnut-mandibled Toucan	22	750	Common

Note that the small icon, image (large icon), and list styles are available only on Windows NT.

A *tree view field* displays hierarchical information in an indented outline, providing controls that let the end user expand and collapse the outline. Unlike an iPlanet UDS outline field, a tree view field has a standardized two-column format, with one small icon column and one text column. The following figure illustrates a tree view field:

**Figure 2-8** Tree View Field

Very often, user interface designers use list view and tree view fields together. A tree view field on the left allows the user to search through a hierarchy for the item he wants, and a list view field on the right provides the detail information for the selected item in the tree view field. The following figure illustrates:

**Figure 2-9** Tree View Field and List View Field Used Together

The following sections provide detailed information about outline fields, list view fields, and tree view fields. For each field, there is a section that provides background information, a section about how to create the field using the Window Workshop, and a section about how to provide the data for the field.

# Using Outline Fields

An outline field provides a browser for multi-column data. It can display a hierarchy of data as an indented outline, or it can display one level of multi-column information. If the data structure is larger than the outline field, the outline field provides a scrollbar that lets the end user scroll through the data.

On Windows NT, list view fields and tree view fields use the window systems' native list view and tree view controls and use a standard format. An outline field, however, is a widget created by iPlanet UDS. You can therefore use it to create a customized widget that displays information in any format you choose.

A tree view field displays information in a hierarchical format, however, you are limited to two standard columns, one icon and one label. A list view field displays any number of columns, however, the list view field provides a flat list, without showing hierarchical relationships. An outline field has none of the restrictions of either the tree view or list view fields, and is therefore useful when you wish to display multiple columns of hierarchical information. Generally, if a list view field or tree view field provides the functionality that you need, you should use those fields. When you need more flexibility, use an outline field.

## Interacting with Outline Fields

The end user interacts with an outline field by:

- single-clicking an item to select it
- double-clicking an item to open it

When controls are turned on for the outline field, the end user can use the controls to expand and collapse the outline.

## Data for Outline Fields

The `DisplayNode` class defines a data object specifically for display in an outline field, list view field, or tree view field. The data for an outline field consists of a hierarchy of `DisplayNode` objects. The `DisplayNode` objects in the hierarchy are displayed in the outline field as rows, with each row indented in the outline to indicate the position of the object in the hierarchy. Each column in the outline field corresponds to a single attribute in the `DisplayNode` class or subclass that is the mapped type of the outline field.



The following figure illustrates the data in an outline field:

**Figure 2-10** Data in an Outline Field

IsGreaterThanOrEqual	: BooleanNullable	← Row (one DisplayNode subclass object)
IsLessThan	: BooleanNullable	
IsLessThanOrEqual	: BooleanNullable	
IsNotEqual	: BooleanNullable	
Mod	:	
Multiply	:	
(source1:integer, source2:integer)	: void	
(source1:NumericData, source2:integer)	: void	
(source1:integer, source2:NumericData)	: void	
(source1:NumericData, source2:NumericData)	: void	
SetValue	:	
(source:DataValue)	: void	← Column (one attribute DisplayNode subclass)
(source:string)	: void	
(source:TextData)	: void	
(source:NumericData)	: void	
(source:ui4)	: void	
(source:long)	: void	
(source:double)	: void	
Subtract	:	
Value	: integer	

To provide the data for an outline field, you can use either the `DisplayNode` class or a user-defined subclass of the `DisplayNode` class. Which class you use, the `DisplayNode` class or a user-defined subclass, depends on whether the outline field has one column or has multiple columns. See [“Providing Data for an Outline Field” on page 96](#) for information on providing the data for single- and multi-column outline fields.

## Node Hierarchy

To construct the hierarchy that is displayed in the outline field, you create the `DisplayNode` objects and link them together using node positioning attributes defined in the `GenericNode` class. You then assign the root node of the hierarchy to the mapped attribute for the outline field. For information on how to construct the node hierarchy, see [“Creating and Assigning the Node Hierarchy” on page 98](#).

## Event Handling

You can handle events on individual rows within the outline field. To tell which row in the outline field was the source of the event, you can use the `LocateNode` method defined for the `OutlineField` class, which allows you to determine which row (or node) the cursor was positioned on when an event was posted. In addition, a number of outline field events provide a node parameter, which gives you the current row.

## Outline Field Properties

The Window Workshop provides several properties that let you control the appearance and behavior of the outline field as a whole.

### Providing Controls

By default, the outline field provides controls that allow the end user to expand and collapse the outline. You can suppress the controls by turning off the `Has Controls` property.

### Displaying the Root Node

As described under [“Creating and Assigning the Node Hierarchy” on page 98](#), the data displayed in the outline field is a node hierarchy, branching from a single root node. By default, the root node of the node hierarchy is not displayed in the outline field. If you wish to display it, you can set the `Root Displayed` property on. The root node will then be displayed at the top level of the outline.

### Turning Scrollbars On and Off

The `Has Horizontal Scrollbar` and `Has Vertical Scrollbar` properties let you specify whether or not the outline field provides horizontal and vertical scroll bars. Keep in mind that if you turn both scroll bars off and then information displayed in the outline field exceeds the size of the outline field, the end user has no way to display it.

### Controlling Row Highlights and Scroll Policy

You can control the behavior of an individual row in the outline field when it is selected by the end user. The `Has Row Highlights` property specifies whether or not an individual row in the list is highlighted when it is selected. Highlighting is reverse video.

The Scroll Policy property specifies where the individual row is scrolled when TOOL code selects the row. (The Scroll Policy property does not take effect when the end user selects a row.) The options are:

Scroll Policy Option	Description
Automatic	Automatic scrolling—moves the node into view. This is the default.
Top	Scrolls the selected node to the top of the list view field. On Windows NT, the selected node is kept in view.
Bottom	Scrolls the selected node to the bottom of the list view field. On Windows NT, the selected node is kept in view.
Middle	Scrolls the selected node to the middle of the list view field. On Windows NT, the selected node is kept in view.
No Scrolling	Does not scroll the selected node. On Windows NT, the selected node is kept in view.

## Displaying Column Titles

You can specify whether or not column titles are displayed by using the Has Column Titles property. The three other list view styles do not display column titles. To specify the title for an individual column, you use the Column Title property (see [“Individual Column Properties” on page 107](#) for information).

## Individual Column Properties

Unlike tree view fields, outline fields do not have a standardized appearance. For each column in the outline field, the Window Workshop allows you to set properties that control the appearance and behavior of the individual column.

### Column Content

To display data in the outline field, you must use the Column Name property. This property specifies the attribute name in the DisplayNode class or subclass that maps to the particular column in the list view field. See [“Providing Data for a List View Field” on page 112](#) for information.

For outline fields, you can set the Column Title property to provide a title for an individual column. After entering the column title, you must be sure to turn on the Has Column Title property for the outline field.

## Column State

By default, the items within an individual column in a outline field are visible, but not draggable. To enable end users to drag the items within a given column, you must set the Column State property to Draggable.

## Column Sizing and Alignment

You control the column sizing and alignment by using the Size Policy, Width, and Alignment properties for the column. The Size Policy property specifies how the column is sized: Fixed or Sized to Text (the default). The Width property specifies the maximum number of characters to display in the column. The Alignment property specifies the alignment of the data within the column: Left, Right, Center, or Default (appropriate for data type).

In TOOL, you can set these column properties using the corresponding attributes in the OutlineColumnDesc class. There is one OutlineColumnDesc object for each column in the list view field. For information, see the OutlineColumnDesc class in the Display Library online Help.

## Column Indenting

An outline field can have any number of columns, based on the attributes in the DisplayNode subclass to which it maps. By default, only the first column in the field uses indenting to indicate the position of the row within the hierarchy.

You may wish to turn on indenting for more than one column. For example, if your first column is an icon and your second column is a text label, you may wish to indent both the first and second columns, so the icon and the text are aligned. The First and Last properties on the column allow you to indicate which column is the first to use indenting and which is the last to use indenting. By default, the first column is both the First and Last column to use indenting.

To indicate which column should be the first to use indenting, you can turn on the First property for the column. Any preceding columns will use not use indenting. To indicate which column should be the last to use indenting, you can turn on the Last property for the column. Any succeeding columns will not use indenting. For example, to align the icon and text label so that both are indented, you would turn on the First property for the icon and the Last property for the text label.

## Creating an Outline Field in the Window Workshop

To create an outline field in the Window Workshop, you can use the Widget > New > OutlineField command or the New Outline Field tool.

By default, the mapped type for an outline field is DisplayNode.

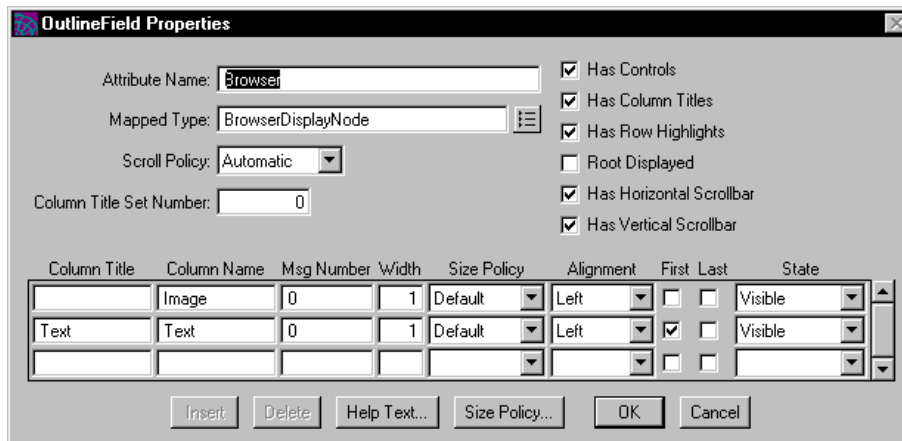
For an outline field with multiple columns, you must use a *subclass* of DisplayNode for the mapped type. The DisplayNode subclass must define the attributes that provide the data for each column you wish to include. The user-defined DisplayNode subclass must be defined before you can completely define the outline field using the Window Workshop. See [“Providing Data for an Outline Field” on page 96](#) for information on defining the DisplayNode subclass.

The OutlineField Properties dialog provides an array field that allows you to define each of the columns in the outline field. The Column Name property in the array field specifies the name of the attribute that defines the particular column. The column name must be an existing attribute in the DisplayNode class (DVNodeText) or subclass (a user-defined attribute). For each Column Name that you enter, iPlanet UDS creates a corresponding column in the outline field. The other properties in the array field allow you to control the appearance and behavior settings for the individual columns.

► **To create an outline field**

1. In the Window Workshop, choose the Widget > New > OutlineField command or click the New Outline Field tool.
2. On the form, draw a rectangle to indicate the size of the outline field you wish to create.

Double-click the outline field to open its properties dialog.



3. On the OutlineField Properties dialog, set the Mapped Type property. Specify either DisplayNode or a user-defined subclass of DisplayNode. Set other outline field properties as desired (see summary below).
4. Define the individual columns. You must enter the Column Name property for each column you wish to display. The other properties are optional.

If you are using DisplayNode, use DVNodeText for the first column in the outline view field. If you are using a user-defined subclass of DisplayNode, you can use DVNodeText for any column in the outline field and use your user-defined attributes for the remaining columns.

The OutlineField Properties dialog allows you to set the following properties for an outline field:

Property	Description
Attribute Name	Sets an attribute name for the outline field.
Mapped Type	Sets the data type for the outline field contents. The mapped type must be the DisplayNode class or a subclass of DisplayNode.

<b>Property</b>	<b>Description</b>
Scroll Policy	Determines where the individual row is scrolled when TOOL code selects the node. (This property does not take effect when the end user selects a node.) Values are: Automatic, Top, Bottom, Middle, and No Scrolling.
Has Controls	Turns the controls for opening and closing folder nodes on or off.
Has Column Titles	Turns the outline field's column titles on or off.
Has Row Highlights	Turns row highlighting for the current node on or off. Highlighting is reverse video.
Root Displayed	Specifies whether or not the root node is displayed.
Has Horizontal Scrollbar	Turns the horizontal scrollbar on or off.
Has Vertical Scrollbar	Turns the vertical scrollbar on or off.
Column Title Set Number	Specifies the set number for the column titles' message numbers. This is for use when creating a multilingual window.
Column Title	Sets the title for an individual column in the outline field.
Column Name	Specifies the name of the attribute in the DisplayNode subclass that defines the data to be displayed in the current column.
Msg Number	Sets the message number for the individual column title. A value of 0 means the current value for the Column Title property. This is for use when creating a multilingual window.
Width	Sets the maximum number of characters that can be displayed in the current column.
Size Policy	Sets the size policy for the current column. Values are Default, Fixed, and Size to Text.
Alignment	Sets the alignment of the data displayed in the field. Values are: Left, Right, and Center.
First/Last	Sets the column as the first or last indent level in the outline field.
State	Sets the state of a column in the outline field: Draggable, Visible, or Invisible.
Insert/Delete	Inserts or deletes a column in the outline field.
Help Text	Opens the Help Text dialog for the field.
Size Policy	Opens the Size Policy dialog for the field.

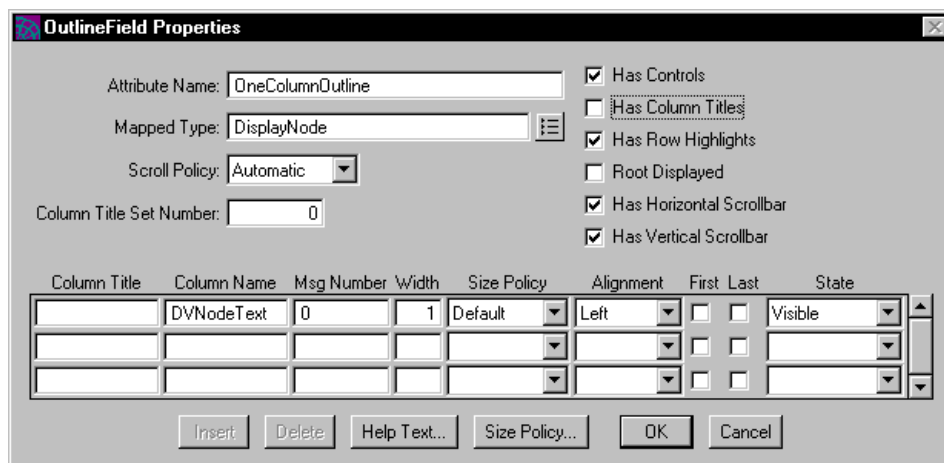
## Providing Data for an Outline Field

To provide the data for an outline field, you can use either the `DisplayNode` class or a user-defined subclass of the `DisplayNode` class. Which class you use, the `DisplayNode` class or a user-defined subclass, depends on whether the outline field has one column or has multiple columns.

If the outline field has only one text column, to provide data for the outline field, you can use the `DVNodeText` attribute of the `DisplayNode` class. You do not need to create a subclass of `DisplayNode`.

To use the `DVNodeText` attribute, in the `OutlineField` properties dialog, specify `DisplayNode` as the mapped type for the outline field and `DVNodeText` as the Column Name Property for the first column in the column list. [Figure 2-11](#) illustrates:

**Figure 2-11** OutlineField Properties Dialog



If the outline field has more than one column, to provide the data for the outline field, you must create a subclass of the `DisplayNode` and, in the new class, define one attribute for each column in the outline field. See [“Using a Subclass of DisplayNode” on page 97](#) for information.

To assign the data to the outline field, you must link the `DisplayNode` subclass objects into a hierarchy and assign the root node of the hierarchy to the mapped attribute of the outline field. See [“Creating and Assigning the Node Hierarchy” on page 98](#) for information.



## Using a Subclass of DisplayNode

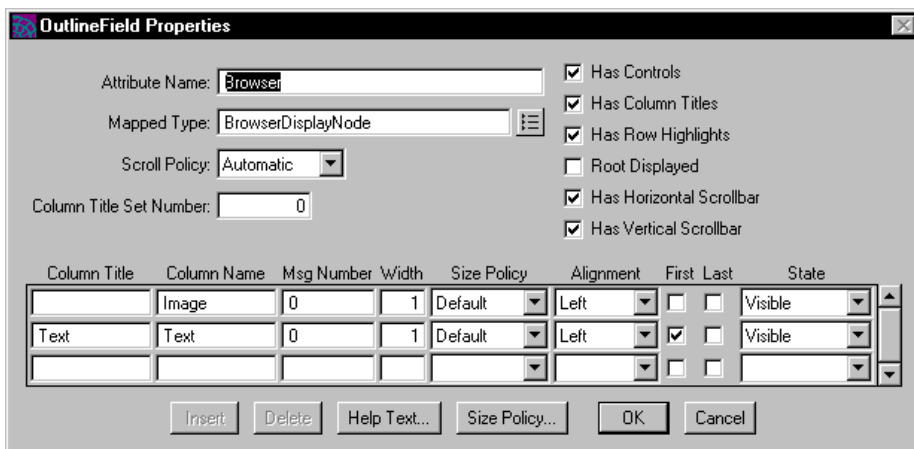
For an outline field that has more than one column, you must create a subclass of the `DisplayNode` class to define the columns to be displayed in the field. The `DisplayNode` subclass defines one attribute for each column that you wish to display. For example, in the iPlanet UDS example program `SimpleOutline`, the `BrowserDisplayNode` class defines two columns to be displayed: a `TextData` attribute and an `ImageData` attribute.

After creating the `DisplayNode` subclass, you must use the Window Workshop to set the mapped type for the field to the `DisplayNode` subclass and map the individual attributes in the subclass to their corresponding columns in the field.

### ► To use the `DisplayNode` subclass

5. In the Project Workshop, create a new class whose superclass is `DisplayNode`.
6. In the Class Workshop, define one attribute for each column you wish to include in the outline field.
7. In the Window Workshop, open the properties dialog for the outline field.
8. In the Properties dialog for the outline field, set the Mapped Type property to the `DisplayNode` subclass.

Also in the Properties dialog, set the Column Name property for each column to the name of the attribute in the `DisplayNode` subclass that provides the data for that particular column.



You are now ready to link the `DisplayNode` objects into a hierarchy and assign the hierarchy to the outline field.

## Creating and Assigning the Node Hierarchy

To link `DisplayNode` objects into a hierarchy, you start by creating a root node. Then, for each additional node that you create, you set the appropriate `DisplayNode` attribute to position the node within the hierarchy.

The *root node* for the node hierarchy is a single `DisplayNode` object that is assigned to the mapped attribute for the field. By default, the root node for the hierarchy is not displayed in the outline field, although you can display it by toggling on the `Root Displayed` property.

The root node must have its `IsFolder` attribute set to `TRUE`. Setting `IsFolder` to `TRUE` allows the root node to have child nodes assigned to it. In addition, you will usually need to set the `IsFilled` and `IsOpened` attributes to `TRUE`. Setting `IsFilled` to `TRUE` specifies that child nodes are loaded and setting `IsOpened` to `TRUE` specifies that the nodes in the folder (the root) are being displayed. You must set these three attributes in the following order:

- `IsFolder`
- `IsFilled`
- `IsOpened`

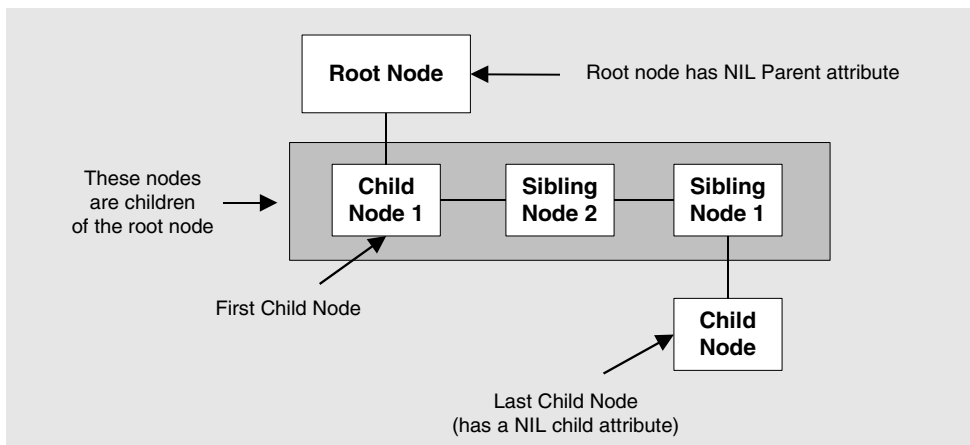
### *Positioning Nodes*

The `GenericNode` class, the superclass of `DisplayNode`, defines the attributes that allow you to position a given node within a hierarchy. (See the Framework Library online Help for complete information on `GenericNode`.) Briefly, the positioning attributes are:

Attribute	Description
Parent	The immediate parent node for the current node. Before you can make another node the parent of the current node, the future parent node's <code>IsFolder</code> attribute must be set to <code>TRUE</code> .
FirstChild	The first child node in the set of immediate child nodes for the current node.
LastChild	The last child node in the set of immediate child nodes for the current node.
NextSibling	The next sibling in the set of siblings for the current node.
PrevSibling	The previous sibling in the set of siblings for the current node.

When you use the Parent attribute to specify a parent for a new node, the new node becomes the last child of the parent node. When you use a “sibling” attribute to specify a sibling for a new node, the new node becomes a child of the sibling’s parent. When you use a “child” attribute to specify a child for a new node, the new node becomes the new parent of the node. The following figure illustrates these relationships:

**Figure 2-12** Nodes in a Node Hierarchy



The following procedure outlines the basic steps for creating the node hierarchy. To provide streamlined code examples, the code used in the procedure illustrates a very simplistic use of outline field with the DisplayNode class as the mapped type. For more sophisticated (and perhaps realistic) examples of outline fields, see the iPlanet UDS example programs SimpleOutline and FileBrowser.

➤ **To create the node hierarchy**

1. Create a local variable and assign to it a new object of the DisplayNode class or subclass. This is your root node. The following code illustrates:

```
-- Initialize a local DisplayNode variable, which we
-- will later assign to the mapped attribute
-- for the Outline field. It is more
-- efficient to build up a local variable and
-- assign it to the mapped attribute, than to directly
-- build the nodes in the mapped attribute.
-- This will become the root node of the Outline.
ourBirdOutline : DisplayNode = new;
```

2. Set the `IsFolder`, `IsFilled`, and `IsOpened` attributes for the root node to `TRUE`.

```
-- Always set the IsFolder, IsFilled, and IsOpened attributes
-- to TRUE in the root node. Always set them in this order.
ourBirdOutline.IsFolder = TRUE;
ourBirdOutline.IsFilled = TRUE;
ourBirdOutline.IsOpened = TRUE;

ourBirdOutline.DVNodeText = 'ROOT NAME';
```

3. Write a loop to create the rest of the nodes. For each `DisplayNode` object you create, use one of the positioning attributes to assign it to its place within the hierarchy.

```
-- We add three levels to the ourBirdOutline
-- DisplayNode. We load them with data from
-- the OrderList array, which is populated in
-- the Init method of this class. This is an
-- appropriate technique when you have small
-- amounts of data. An alternative technique,
-- if you have a lot of data to display, would
-- be to create the first level of the hierarchy
-- only, connect each node at this level to the data
-- in the next level down with DisplayNode's Related
-- attribute, and build out the lower levels at
-- runtime. Note that when you use the alternate
-- technique, you should always set the IsFolder
-- attribute to TRUE, regardless of whether the node
-- has a child node. Doing so will allow the user to
-- click the control, generating an event which you can
-- handle to build the child node.
```

```
Node : DisplayNode;
ChildNode : DisplayNode;
GrandChildNode : DisplayNode;

for order in OrderList do
  Node = new;
  Node.DVNodeText = new;
  Node.DVNodeText.SetValue(order.Name);
  Node.IsFolder = FALSE;
  Node.IsFilled = TRUE;
  Node.IsOpened = TRUE;
  -- Parent 'Order' level node to the root node.
  Node.Parent = ourBirdOutline;

  if order.FamilyList <> nil then
    Node.IsFolder = TRUE;
    for family in (order.FamilyList) do
```

```

ChildNode = new;
ChildNode.DVNodeText = new;
ChildNode.DVNodeText.SetValue(family.Name);
ChildNode.IsFolder = FALSE;
ChildNode.IsFilled = TRUE;
ChildNode.IsOpened = FALSE;
-- Parent the 'Family' level node to the 'Order' level node.
ChildNode.Parent = Node;

if family.GenusList <> nil then
  ChildNode.IsFolder = TRUE;
  for genus in family.GenusList do
    GrandChildNode = new;
    GrandChildNode.DVNodeText = genus.Name;
    GrandChildNode.IsFolder = FALSE;
    GrandChildNode.IsFilled = TRUE;
    GrandChildNode.IsOpened = FALSE;
    -- Parent 'Genus' level node to the 'Family' level node.
    GrandChildNode.Parent = ChildNode;

    -- If genus has species data associated with it, we will
    -- display species information in the ListView field,
    -- when a user clicks on that genus. For now, we use the
    -- Related attribute to connect a node with its species
    -- list. We will build species display node when it's
    -- requested.
    if genus.SpeciesList <> nil then

      ourGenusInfo : GenusInfo = new;
      ourGenusInfo.SpeciesList = genus.SpeciesList;
      GrandChildNode.Related = ourGenusInfo;

    end if;
  end for;
end if;

end for;
end if;

end for;

```

4. When the node hierarchy is complete, assign the local variable for the root node to the mapped attribute for the outline field.

```

-- Assign the local variable to the mapped attribute, now that the
-- structure is complete.
BirdOutline = ourBirdOutline;

```

## Using List View Fields

As described above, the list view field displays a set of items, each consisting of an icon and a text label. On Windows NT, iPlanet UDS uses the native list view control. On all other window systems, iPlanet UDS creates a custom widget.

The `ListView` class, which defines the list view widget, is a subclass of the `OutlineField` class, and it provides a simplified version of the outline field. The advantages of using a list view field over an outline field are:

- iPlanet UDS provides the look and feel of a list view control
- it is easier to define the data

The `DisplayNode` class provides attributes that define the first two columns of the list view field, `DVSmallIcon`, `DVLargeIcon`, and `DVNodeText`. For the small icon, image (large icon), and list styles, and for some detail list view fields, these attributes are all you need.

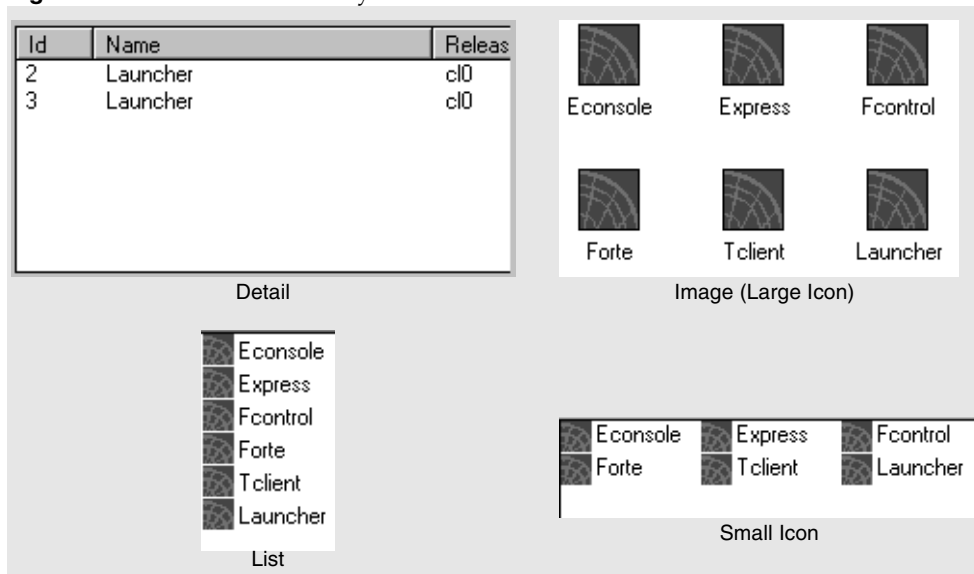
- it is easier to assign the data

You do not need to create a node hierarchy for a list view field as you do with an outline field. Instead, you can simply assign an array of nodes to the list view field.

- on Windows NT, the native list view control provides automatic column sorting and resizable columns

## List View Styles

When you create a list view field, you can choose one of four styles. For the small icon, image (large icon), and list styles, there are always two standard columns, one icon and one text column. (Figure 2-13 shows how each of these styles displays the two “columns.”) For the detail style, you can create any number of resizable columns. The following figure illustrates the list view field styles:

**Figure 2-13** List View Field Styles

The Detail example in [Figure 2-13](#) shows no icons because none were defined. All styles can display icons if they are defined in the `DisplayNode` array, as described in [“Providing Data for a List View Field”](#) on page 112.

## Styles and Portability

The small icon, image (large icon), and list styles are available on Windows NT only. Therefore, if you plan to deploy your application on more than one window system, we recommend that you use only the detail style. Using the detail style ensures visual consistency between the various platforms.

## Styles and Sorting

On Windows NT, the detail list view field provides automatic sorting, in ascending order, when the end user clicks on the list view field’s header. If you wish to provide a descending sort, you can handle the list view field’s `RequestSort` event (see the Display Library online Help for information).

The following sections provide detailed information about how end users interact with list view fields, about how you provide the data for the list view field, and about the properties of a list view field.

## Interacting with List View Fields

The end user interacts with a list view field by single or double-clicking the items in the list. Your application controls what happens in response to these clicks.

On Windows NT, detail style list view fields provide the following special features:

- the end user can click in the header area to sort the items in the list
- the end user can resize the columns by dragging the column boundaries

## Data for List View Fields

The `DisplayNode` class defines a data object specifically for display in an outline field, list view field, or tree view field. The data for a list view field consists of an array of `DisplayNode` objects. The `DisplayNode` class defines a single data node in the list view field. Each item or “row” in the list view field corresponds one to one with a `DisplayNode` object.

The following figure illustrates the data in a list view field:

**Figure 2-14** Data for List View Field

Species	Length	Weight	Status
Keel-billed Toucan	18	500	Common
Chestnut-mandibled Toucan	22	750	Common

Row (one `DisplayNode` subclass object)

Column (one attribute of `DisplayNode` subclass)



To provide the data for a list view field, you construct an array of `DisplayNode` objects (or `DisplayNode` subclass objects) and assign it to the list view field using the `SetViewNodes` method. See [“Providing Data for a List View Field” on page 112](#) for information about constructing and assigning the array.

Note that you can display a node hierarchy (rather than a node array) in the list view field the same way you do in a tree view field. However, only the immediate children of the root node will be displayed in the list view field. Other nodes lower in the hierarchy will be ignored. See [“Providing Data for a Tree View Field” on page 121](#) for information on assigning a node hierarchy to the field.

You handle events on the list view field the same way you do with outline fields. To tell which particular node in the field was the source of the event, you can use the `LocateNode` method defined for the `OutlineField` class, which allows you to determine which node the cursor was positioned on when an event was posted. A number of list view field events provide a node parameter, which gives you the current node.

## List View Properties

The Window Workshop provides several properties that let you control the appearance and behavior of the list view field as a whole.

### List View Style

The default style for a list view field is the detail list view style. For the Windows NT, you can set the style to the following settings: small icon, image (large icon), or list. On all other platforms, iPlanet UDS uses the detail style, no matter which list view style setting you specify.

### Scrollbars

The `Has Horizontal Scrollbar` and `Has Vertical Scrollbar` properties let you specify whether or not the list view field provides horizontal and vertical scroll bars. On platforms other than Windows NT, you control the horizontal and vertical scrollbars individually.

On Windows NT, the window system provides scroll bars only when they are necessary. In addition, you cannot control the horizontal and vertical scrollbars individually. Therefore, if you turn on either `Has Horizontal Scrollbar` or `Has Vertical Scrollbar`, Windows NT displays *both* scroll bars, but only when the information in the field exceeds the size of the field. To turn off the scrollbars, you must set both `Has Horizontal Scrollbar` and `Has Vertical Scrollbar` off.

Keep in mind that if you turn both scroll bars off and then information displayed in the list view field exceeds the size of the list view field, the end user has no way to display it.

## Row Highlights and Scroll Policy

You can control the behavior of an individual row in the list view field when it is selected by the end user. The Has Row Highlights property specifies whether or not an individual row in the list is highlighted when it is selected. Highlighting is reverse video. Note that row highlighting is not available for Windows NT.

The Scroll Policy property specifies where the individual row is scrolled when it is selected. The options are:

Scroll Policy Option	Description
Automatic	Automatic scrolling—moves the node into view. This is the default.
Top	Scrolls the selected node to the top of the list view field. On Windows NT, the selected node is kept in view.
Bottom	Scrolls the selected node to the bottom of the list view field. On Windows NT, the selected node is kept in view.
Middle	Scrolls the selected node to the middle of the list view field. On Windows NT, the selected node is kept in view.
No Scroll	Does not scroll the selected node. On Windows NT, the selected node is kept in view.

## Column Titles

For detail list view fields, you can specify whether or not column titles are displayed by using the Has Column Titles property. The three other list view styles do not display column titles. To specify the title for an individual column, you use the Column Title property (see [“Individual Column Properties” on page 107](#) for information).

## Individual Column Properties

Unlike tree view fields, list view fields do not have a standardized appearance. For each column in the list view field, the Window Workshop allows you to set properties that control the appearance and behavior of the individual column.

### Column Content

To display data in the list view field, you must use the Column Name property. This property specifies the attribute name in the `DisplayNode` class or subclass that maps to the particular column in the list view field. See [“Providing Data for a List View Field” on page 112](#) for information.

For detail list view fields, you can set the Column Title property to provide a title for an individual column. After entering the column title, you must be sure to turn on the Has Column Title property for the list view field. For list view styles other than detail, the column titles are *not* displayed, even if you enter the titles and turn Has Column Titles on.

### Column State

By default, the items within an individual column in a list view field are visible, but not draggable. To enable end users to drag the items within a given column, you must set the Column State property to Draggable.

### Column Sizing and Alignment

You control the column sizing and alignment by using the Size Policy, Width, and Alignment properties for the column. The Size Policy property specifies how the column is sized: Fixed or Sized to Text (the default). The Width property specifies the maximum number of characters to display in the column. The Alignment property specifies the alignment of the data within the column: Left, Right, Center, or Default (appropriate for data type).

In TOOL, you can set these column properties using the corresponding attributes in the `OutlineColumnDesc` class. There is one `OutlineColumnDesc` object for each column in the list view field. For information, see the `OutlineColumnDesc` class in the Display Library online Help.

## Creating a List View Field in the Window Workshop

To create a list view field in the Window Workshop, you can use the Widget > New > ListView command or the New List View Field tool.

### Setting List View Style

The New > ListView command creates a detail list view field. For Windows NT, if you want to use a different list view style, the ListView Properties dialog provides a List Style property, which allows you to choose Small Icon, Image (large icon), or List. Keep in mind, however, on all window systems except Windows NT, the only style that is available is the detail style. Therefore, even if you choose a different style, the list view field will be displayed in the detail style on platforms other than Windows NT.

By default, the mapped type for a list view field is DisplayNode. You can use the DisplayNode class as the mapped type for the following list view styles:

- small icon style list view field
- image (large icon) style list view field
- list style list view field
- detail style list view field with only one text column

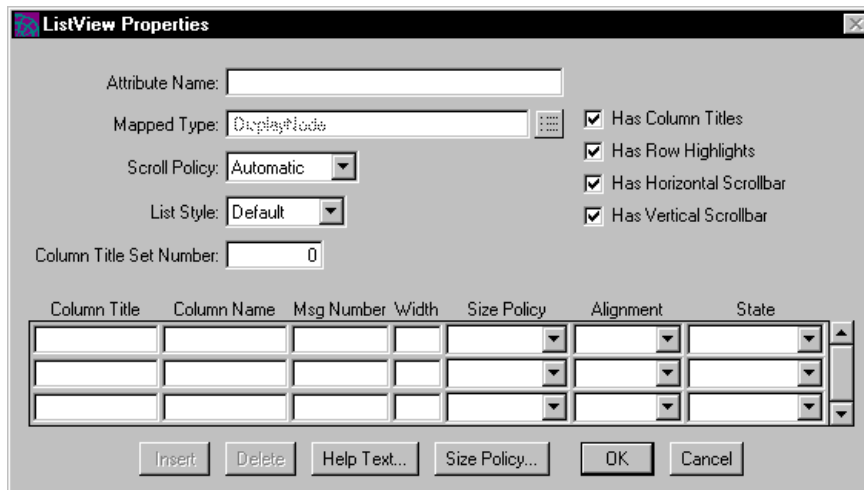
For a detail list view field with multiple columns, you must use a *subclass* of DisplayNode for the mapped type. The DisplayNode subclass must define the attributes that provide the data for each column you wish to include. The user-defined DisplayNode subclass must be defined before you can completely define the list view field using the Window Workshop. See [“Providing Data for a List View Field” on page 112](#) for information on defining the DisplayNode subclass.

### Column Names and Other Column Properties

The ListView Properties dialog provides an array field that allows you to define each of the columns in the list view field. The Column Name property in the array field specifies the name of the attribute that defines the particular column. The column name must be an existing attribute in the DisplayNode class (DVNodeText) or subclass (a user-defined attribute). For each Column Name that you enter, iPlanet UDS creates a corresponding column in the list view field. The other properties in the array field allow you to control the appearance and behavior settings for the individual columns.

► **To create a list view field**

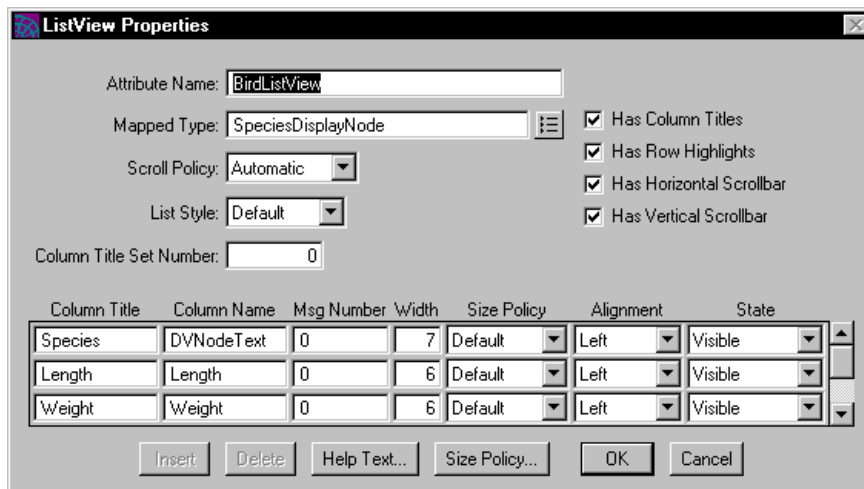
1. In the Window Workshop, choose the Widget > New > ListView command or click the New List View tool.
2. On the form, draw a rectangle to indicate the size of the list view field you wish to create.
3. Double-click the list view field to open its properties dialog.



4. On the ListView Properties dialog, use the List Style property to set the style of the list view field. (Small Icon, Image (large icon), and List styles take effect only on Windows NT.)
5. For the Mapped Type property, specify either DisplayNode or a user-defined subclass of DisplayNode. Set other list view field properties as desired (see summary below).

- Define the individual columns. You must enter the Column Name property for each column you wish to display. The other properties are optional.

If you are using `DisplayNode`, use `DVNodeText` for the first column name in the list view field. If you are using a user-defined subclass of `DisplayNode`, you can use `DVNodeText` for any column in the list view field and use your user-defined attributes for the remaining columns.



The List View Properties dialog, shown the figure above, allows you to set the following properties for a list view field:

Property	Description
Attribute Name	Sets an attribute name for the list view field.
Mapped Type	Sets the data type for the list view field contents. The type must be the <code>DisplayNode</code> class or a user-defined subclass of <code>DisplayNode</code> .
Scroll Policy	Determines where list view field scrolls a node when it becomes the current node. Values are: Automatic, Top, Bottom, Middle, and No Scroll.
List Style	Sets the style of the list view field: Detail, Image (large icon), Small Icon, and List. Image, Small Icon, and List styles are available only on Windows NT.

<b>Property</b>	<b>Description</b>
Column Title Set Number	Sets number for the column titles' message numbers. This is for use when internationalizing the application.
Has Column Titles	Turns the list view field's column titles on or off.
Has Row Highlights	Sets row highlighting for the current node in the list view field. Highlighting is reverse video.
Has Horizontal Scrollbar	Turns the horizontal scroll bar on or off.
Has Vertical Scrollbar	Turns the vertical scroll bar on or off.
Column Title	Sets the title for an individual column in the list view field.
Column Name	Specifies the name of the attribute in the DisplayNode class (or subclass) that defines the data to be displayed in the current column. The attribute is either the DVNodeText attribute or a user-defined attribute in the DisplayNode subclass (the mapped type).
Msg Number	Specifies the message number for the individual column title. A value of 0 means the current value for the Column Title property. This is for use when internationalizing the application.
Width	Sets the maximum number of characters that can be displayed in the current column.
Size Policy	Sets the size policy for the current column. Values are Default, Fixed, and Size to Text.
Alignment	Sets the alignment of the data displayed in the field. Values are: Default (appropriate for the type), Left, Right, and Center.
State	Sets the state of a column in the list view field: Default, Draggable, Visible, or Invisible.
Insert/Delete	Inserts or deletes a column in the list view field.
Help Text	Opens the Help Text dialog for the field.
Size Policy	Opens the Size Policy dialog for the field.

## Providing Data for a List View Field

The way you provide data for list view fields depends on the list view field style. For small icon, image, list, and detail list view fields with a single text column, you can use the `DisplayNode` class without subclassing. For detail list view fields with multiple columns, you must create a subclass of the `DisplayNode` class.

### Small Icon and Simple List Fields

For the small icon and simple list view fields, the `DVSmallIcon` attribute in the `DisplayNode` class defines the icon column and the `DVNodeText` attribute in the `DisplayNode` class defines the text column.

To use these attributes, in the `ListView` properties dialog, specify `DisplayNode` as the mapped type for the list view field and `DVNodeText` as the Column Name Property for the first column in the column list.

### Image List View Field

For the image (large icon) list view field, the `DVLargeIcon` attribute in the `DisplayNode` class contains the large icon and the `DVNodeText` attribute in the `DisplayNode` class contains the text label for the icon. As shown above for small icon and simple list fields, you must specify `DisplayNode` as the mapped type for the list view field and `DVNodeText` as the Column Name Property for the first column in the column list.

### Detail List View Field

For a detail list view that displays a single text column, you can use the `DisplayNode` class as described above for the small icon and simple list views. However, for a detail list view field that displays multiple columns, you must create a subclass of the `DisplayNode` class to define the individual columns in the list. In the `DisplayNode` subclass, you must create one attribute for each column in the list. See [“Using a Subclass of DisplayNode” on page 113](#) for information.

### Using a DisplayNode Array

A list view always displays information in a non-hierarchical format. Therefore, it is not necessary for you to construct a node hierarchy the way you must with outline fields and tree view fields. All you need to do is to create an array of the `DisplayNode` objects (or `DisplayNode` subclass objects) and assign that array to the list view field using the `SetViewNodes` method defined for the `ListView` class. See [“Creating and Assigning the Node Array” on page 114](#) for information.



If you wish to assign a node hierarchy to the list view field, you can do so as described under [“Providing Data for a Tree View Field” on page 121](#). However, only the immediate children of the root node will be displayed in the list view field. Other nodes lower in the hierarchy will be ignored. Using the `SetViewNodes` method is much easier.

## Using a Subclass of `DisplayNode`

For detail list view fields that have more than one column, you must create a subclass of the `DisplayNode` class to define the columns to be displayed in the field.

For detail list view fields, the `DisplayNode` subclass provides one automatic column for the list, the icon column. The `DVSmallIcon` attribute, inherited from the `DisplayNode` class, defines this icon column. In addition, you can use the `DVNodeText` attribute, inherited from the `DisplayNode` class, to define one of the text columns. You must create new attributes for the remaining columns you wish to display.

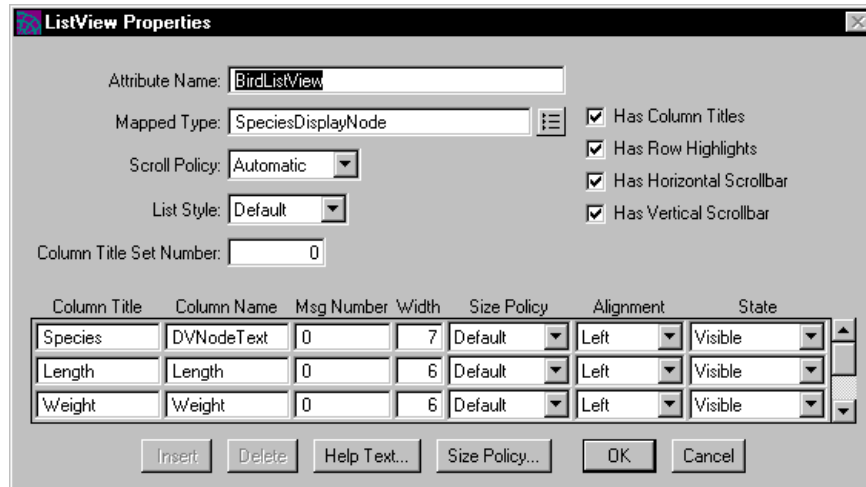
After creating the `DisplayNode` subclass, you must use the Window Workshop to set the mapped type for the field to the `DisplayNode` subclass and map the individual attributes in the subclass to their corresponding columns in the field.

### ► To use the `DisplayNode` subclass

1. In the Project Workshop, create a new class whose superclass is `DisplayNode`.
2. In the Class Workshop, define one attribute for each column you wish to include in the list view field.
3. In the Window Workshop, open the properties dialog for the list view field.

- In the Properties dialog for the list view field, set the Mapped Type property to the `DisplayNode` subclass.

Also in the Properties dialog, set the Column Name property for each column to the name of the attribute in the `DisplayNode` subclass that provides the data for that particular column.



You are now ready to create the array of `DisplayNode` subclass objects and use the `SetViewNodes` method to assign the array to the list view field.

## Creating and Assigning the Node Array

To construct the node array, simply create an array of `DisplayNode` or of your user-defined `DisplayNode` subclass. Then, create the `DisplayNode` objects in the array, in the order in which you wish to display them. (The order of the rows in the array determines the order in which the nodes are displayed in the list view field.) The following code illustrates:

The following code illustrates:

```
-- In this example, SpeciesList is an array of SpeciesInfo.
-- SpeciesInfo is a small class, with four attributes:
-- Name, Length, Weight, and Status. SpeciesDisplayNode is a
-- subclass of DisplayNode with three additional attributes:
-- Length, Weight, and Status. SpeciesList has been initialized
-- and assigned values in the Init method.

-- Loop through the SpeciesList array, and assign its data
-- to our SpeciesDisplayNode.
i : integer = 1;
ourSpeciesNodeList : array of SpeciesDisplayNode = new;
```

```

while i <= SpeciesList.Items do
    ourSpeciesNodeList[i] = new;
    ourSpeciesNodeList[i].DVNodeText = new;

    ourSpeciesNodeList[i].DVNodeText.SetValue(SpeciesList[i].Name);
    ourSpeciesNodeList[i].Length = SpeciesList[i].Length;
    ourSpeciesNodeList[i].Weight = SpeciesList[i].Weight;
    ourSpeciesNodeList[i].Status = SpeciesList[i].Status;
    i = i + 1;
end while;

```

**Project:** TreeList • **Class:** MainWindow • **Method:** Display

After creating the node array, you assign it to the list view field using the `SetViewNodes` method defined for the `ListView` class. The `SetViewNodes` method has a single parameter, `columns`, which provides a reference to the node array. (See the Display Library online Help for information on the `SetViewNodes` method.) The following code illustrates using the `SetViewNodes` method:

```

-- Populate the ListView field with the data in the array of your
-- subclass of DisplayNode.
<BirdListView>.SetViewNodes(ourSpeciesNodeList);

```

## Using Tree View Fields

As described earlier, the tree view field displays hierarchical information in standardized two-column, indented outline format. On Windows NT, iPlanet UDS uses the native list view control. On all other window systems, iPlanet UDS creates a custom widget.

The `TreeView` class, which defines the tree view widget, is a subclass of the `OutlineField` class, and it provides a simplified version of the outline field. The advantages of using a tree view field over an outline field are:

- iPlanet UDS provides the look and feel of a tree view control
- it is easier to define the data

The `DisplayNode` class provides attributes that define both columns of the tree view field. These attributes are all you need.

The following sections provide detailed information about how end users interact with tree view fields, about how you provide the data for the tree view field, and about the properties of a tree view field.

## Interacting with Tree View Fields

The end user interacts with a tree view field by:

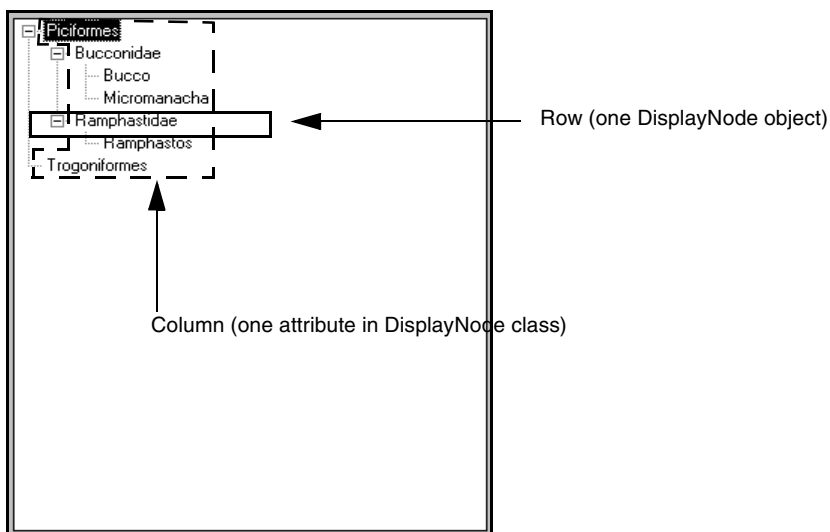
- single-clicking an item to select it
- double-clicking an item to open it

When controls are turned on for the tree view field, the end user can use the controls to expand and collapse the outline.

## Data for Tree View Fields

The `DisplayNode` class defines a data object specifically for display in an outline field, list view field, or tree view field. The data for a tree view field, like an outline field, consists of a hierarchy of `DisplayNode` objects. The `DisplayNode` objects in the hierarchy are displayed in the tree view field as rows, with each row indented to indicate the position of the object in the hierarchy.

The following figure illustrates the data in a tree view field:

**Figure 2-15** Data in a Tree View Field

Unlike an outline field, a tree view field always displays hierarchical information in a standard format, with two standard columns, a small icon column and a text data column. Therefore, the `DisplayNode` class provides the attributes you need for these two columns: `DVSmallIcon`, `DVSelectedIcon`, and `DVNodeText`. You do not need to create a subclass of `DisplayNode`.

### DVSmallIcon and DVSelectedIcon Attributes

The `DVSmallIcon` and `DVSelectedIcon` attributes set the icons that are displayed in the first column. The `DVSmallIcon` attribute sets the icon that is displayed when the node is unselected. The `DVSelectedIcon` sets the icon that is displayed when the node is selected.

### DVNodeText Attribute

The `DVNodeText` attribute sets the text that is displayed in the second column.

### Node Hierarchy

To construct the node hierarchy, you create the `DisplayNode` objects and link them together using node positioning attributes defined in the `GenericNode` class. You then assign the root node of the hierarchy to the mapped attribute for the tree view field. See [“Providing Data for a Tree View Field” on page 121](#) for information on how to construct the node hierarchy for a tree view field.

## Event Handling

You handle events on the tree view field the same way you do with outline fields. To tell which particular node in the field was the source of the event, you can use the `LocateNode` method defined for the `OutlineField` class, which allows you to determine which node the cursor was positioned on when an event was posted. A number of tree view field events provide a parameter which gives you the current node. The iPlanet UDS example program `TreeList` uses the node parameter on the `AfterCurrentNodeChange` event to access the current node.

## Tree View Properties

The Window Workshop provides several properties that let you control the appearance and behavior of the tree view field as a whole.

### Providing Controls

By default, the tree view field provides controls that allow the end user to expand and collapse the outline. You can suppress the controls by turning off the `Has Controls` property.

### Allowing Dragging

You may or may not want your end users to be able to drag the individual nodes in the tree view field. By default, the nodes in the tree view field are visible, but not draggable. To allow the end user to drag the nodes, turn on the `Is Draggable` property.

### Displaying the Root Node

As described under [“Providing Data for a Tree View Field” on page 121](#), the data displayed in tree view field is a node hierarchy, branching from a single root node. By default, the root node of the node hierarchy is not displayed in the tree view field. If you wish to display it, you can set the `Root Displayed` property on. The root node will then be displayed at the top level of the outline.

## Turning Scroll Bars On and Off

The Has Horizontal Scrollbar and Has Vertical Scrollbar properties let you specify whether or not the tree view field provides horizontal and vertical scroll bars. On platforms other than Windows NT, you control the horizontal and vertical scrollbars individually.

On Windows NT, the window system provides scroll bars only when they are necessary. In addition, you cannot control the horizontal and vertical scrollbars individually. Therefore, if you turn on either Has Horizontal Scrollbar or Has Vertical Scrollbar, Windows NT displays *both* scroll bars, but only when the information in the field exceeds the size of the field. To turn off the scrollbars, you must set both Has Horizontal Scrollbar and Has Vertical Scrollbar off.

Keep in mind that if you turn both scroll bars off and then information displayed in the tree view field exceeds the size of the tree view field, the end user has no way to display it.

## Row Highlights and Scroll Policy

You can control the behavior of an individual row in the tree view field when it is selected by the end user. The Has Row Highlights property specifies whether or not an individual row in the list is highlighted when it is selected. Highlighting is reverse video. Note that row highlighting is not available for Windows NT.

The Scroll Policy property specifies where the individual row is scrolled when it is selected. The options are:

Scroll Policy Option	Description
Automatic	Automatic scrolling—moves the node into view. This is the default.
Top	Scrolls the selected node to the top of the tree view field.
Bottom	Scrolls the selected node to the bottom of the tree view field.
Middle	Scrolls the selected node the middle of the tree view field.
No Scroll	Does not scroll the selected node.

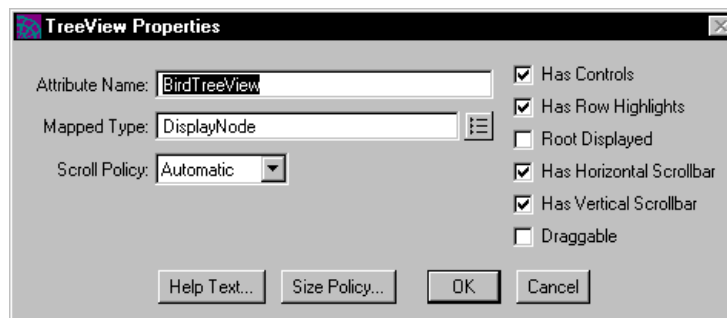
## Creating a Tree View Field in the Window Workshop

To create a tree view field in the Window Workshop, you can use the Widget > New > TreeView command or the New Tree View Field tool.

### ► To create a tree view field

1. In the Window Workshop, choose the Widget > New > TreeView command or click the New Tree View tool.
2. On the form, draw a rectangle to indicate the size of the tree view field you wish to create.
3. Double-click the tree view field to open its properties dialog.

On the TreeView Properties dialog, set the tree view field properties as desired (see summary below).



The following table describes the tree view field properties:

Property	Description
Attribute Name	Sets an attribute name for the tree view field.
Mapped Type	Sets the data type for the tree view field contents. The mapped type must be the DisplayNode class or a subclass of DisplayNode.
Scroll Policy	Determines where list view field scrolls a node when it becomes the current node. Values are: Automatic, Top, Bottom, Middle, and No Scroll.
Has Controls	Turns the controls for opening and closing folder nodes on or off.
Has Row Highlights	Turns row highlighting for the current node on or off. Highlighting is reverse video.



Property	Description
Root Displayed	Specifies whether or not the root node is displayed.
Has Horizontal Scrollbar	Turns the horizontal scroll bar on or off.
Has Vertical Scrollbar	Turns the vertical scroll bar on or off.
Is Draggable	Allows the end user to drag the individual nodes in the tree view field.
Help Text	Opens the Help Text dialog for the field.
Size Policy	Opens the Size Policy dialog for the field.

## Providing Data for a Tree View Field

As described under [“Data for Tree View Fields” on page 116](#), the data for a tree view field consists of a hierarchy of `DisplayNode` objects.

To link `DisplayNode` objects into a hierarchy, you start by creating a root node. Then, for each additional node that you create, you set the appropriate `DisplayNode` attribute to position the node within the hierarchy.

The *root node* for the node hierarchy is a single `DisplayNode` object that is assigned to the mapped attribute for the field. By default, the root node for the hierarchy is not displayed in the tree view field or outline field, although you can display it by toggling on the `Root Displayed` property.

The root node must have its `IsFolder` attribute set to `TRUE`. Setting `IsFolder` to `TRUE` allows the root node to have child nodes assigned to it. In addition, you will usually need to set the `IsFilled` and `IsOpened` attributes to `TRUE`. Setting `IsFilled` to `TRUE` specifies that child nodes are loaded and setting `IsOpened` to `TRUE` specifies that the nodes in the folder (the root) are being displayed. You must set these three attributes in the following order:

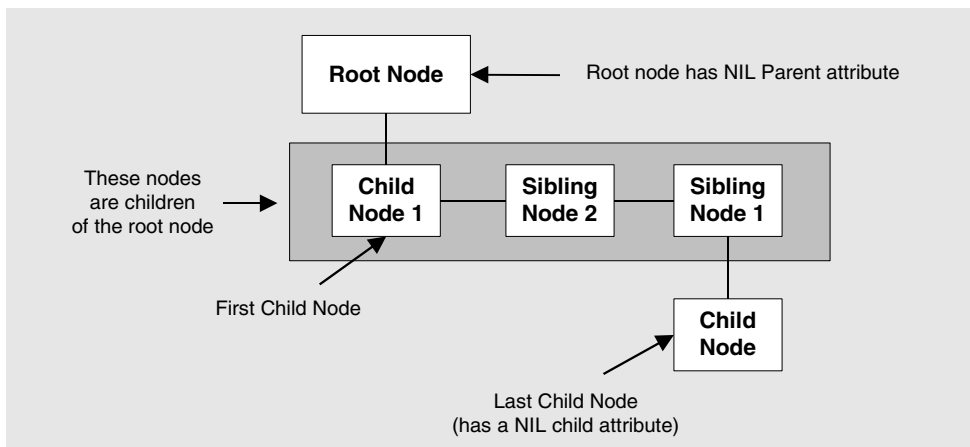
- `IsFolder`
- `IsFilled`
- `IsOpened`

## Attributes for Positioning Nodes

The `GenericNode` class, the superclass of `DisplayNode`, defines the attributes that allow you to position a given node within a hierarchy. (See the Framework Library online Help for complete information on `GenericNode`.) Briefly, the positioning attributes are:

Attribute	Description
Parent	The immediate parent node for the current node. Before you can make another node the parent of the current node, the future parent node's <code>IsFolder</code> attribute must be set to <code>TRUE</code> .
FirstChild	The first child node in the set of immediate child nodes for the current node.
LastChild	The last child node in the set of immediate child nodes for the current node.
NextSibling	The next sibling in the set of siblings for the current node.
PrevSibling	The previous sibling in the set of siblings for the current node.

When you use the `Parent` attribute to specify a parent for a new node, the new node becomes the last child of the parent node. When you use a "sibling" attribute to specify a sibling for a new node, the new node becomes a child of the sibling's parent. When you use a "child" attribute to specify a child for a new node, the new node becomes the new parent of the node. The following figure illustrates these relationships:

**Figure 2-16** Nodes in a Node Hierarchy

► **To create the node hierarchy**

1. Create a local variable and assign to it a new object of the `DisplayNode` class or subclass. This is your root node. The following code illustrates:

```
-- Initialize a local DisplayNode variable, which we
-- will later assign to the mapped attribute
-- for the TreeView field. It is more
-- efficient to build up a local variable and
-- assign it to the mapped attribute, than to directly
-- build the nodes in the mapped attribute.
ourBirdTreeView : DisplayNode = new;
```

2. Set the `IsFolder`, `IsFilled`, and `IsOpened` attributes for the root node to `TRUE`.

```
-- Always set the IsFolder, IsFilled, and IsOpened attributes
-- to TRUE in the root node. Always set them in this order.
-- This will become the root node of the TreeView.
ourBirdTreeView.IsFolder = TRUE;
ourBirdTreeView.IsFilled = TRUE;
ourBirdTreeView.IsOpened = TRUE;

ourBirdTreeView.DVNodeText = 'ROOT NAME';
```

- Write a loop to create the rest of the nodes. For each `DisplayNode` object you create, use one of the positioning attributes to assign it to its place within the hierarchy.

```
-- We add three levels to the ourBirdTreeView
-- DisplayNode. We load them with data from
-- the OrderList array, which is populated in
-- the Init method of this class. This is an
-- appropriate technique when you have small
-- amounts of data. An alternative technique,
-- if you have a lot of data to display, would
-- be to create the first level of the hierarchy
-- only, connect each node at this level to the data
-- in the next level down with DisplayNode's Related
-- attribute, and build out the lower levels at
-- runtime. Note that when you use the alternate
-- technique, you should always set the IsFolder
-- attribute to TRUE, regardless of whether the node
-- has a child node. Doing so will allow the user to
-- click the control, generating an event which you can
-- handle to build the child node.

Node : DisplayNode;
ChildNode : DisplayNode;
GrandChildNode : DisplayNode;

for order in OrderList do
  Node = new;
  Node.DVNodeText = new;
  Node.DVNodeText.SetValue(order.Name);
  Node.IsFolder = FALSE;
  Node.IsFilled = TRUE;
  Node.IsOpened = TRUE;

  -- Parent 'Order' level node to the root node.
  Node.Parent = ourBirdTreeView;

  if order.FamilyList <> nil then
    Node.IsFolder = TRUE;
    for family in (order.FamilyList) do
      ChildNode = new;
      ChildNode.DVNodeText = new;
      ChildNode.DVNodeText.SetValue(family.Name);
      ChildNode.IsFolder = FALSE;
      ChildNode.IsFilled = TRUE;
      ChildNode.IsOpened = FALSE;

      -- Parent the 'Family' level node to the 'Order' level node.
      ChildNode.Parent = Node;

      if family.GenusList <> nil then
        ChildNode.IsFolder = TRUE;
        for genus in family.GenusList do
          GrandChildNode = new;
```

```

GrandChildNode.DVNodeText = genus.Name;
GrandChildNode.IsFolder = FALSE;
GrandChildNode.IsFilled = TRUE;
GrandChildNode.IsOpened = FALSE;

-- Parent 'Genus' level node to 'Family' level node.
GrandChildNode.Parent = ChildNode;

-- If genus has species data associated with it, we will
-- display the species information in ListView field,
-- when a user clicks on that genus. For now, we use the
-- Related attribute to connect this node with its
-- species list. We will build the species display node
-- when it's requested.
if genus.SpeciesList <> nil then
    ourGenusInfo : GenusInfo = new;
    ourGenusInfo.SpeciesList = genus.SpeciesList;
    GrandChildNode.Related = ourGenusInfo;
end if;
end for;
end if;

end for;
end if;

end for;

```

4. When the node hierarchy is complete, assign the local variable for the root node to the mapped attribute for the tree view field.

```

-- Assign the local variable to the mapped attribute, now that the
-- structure is complete.
BirdTreeView = ourBirdTreeView;
-- It's a good practice to do all this before the open.
self.Open();
...

```

**Project:** TreeList • **Class:** MainWindow • **Method:** Display



# Creating a Portable User Interface

This chapter provides background information about the issues you need to consider when creating a portable user interface, and how to use the portability features that iPlanet UDS provides.

In this chapter, you will learn how to:

- use grid fields
- use field partnerships
- use widget sizing properties

For information on how to structure your user interface, see [Chapter 1, “How to Structure a Graphical User Interface.”](#) For complete information on creating tab folders, outline fields, list view fields, and tree view fields, see [Chapter 2, “Using Complex Widgets.”](#)

## Designing a Portable User Interface

All the windows you create using the Window and Menu Workshops will run on any window system that iPlanet UDS supports. However, because of differences between the window systems, the same window may look different from system to system. When you design a user interface that will run on more than one window system, you need to ensure that the interface will look good on all the window systems. To do this, there are several issues you must consider.

The following sections provide information about the window system differences you need to be aware of when designing your user interface and, when appropriate, describes the features that iPlanet UDS provides to help you create a portable user interface.

## Widget Differences

To provide a user interface that has the native look and feel of each window system, iPlanet UDS always uses the widgets (also called controls) provided by the native window toolkit. The advantage of this technique is that a widget always looks the way the end user expects it to.

### Tab Folders

If there are more tabs than can fit across the tab folder, iPlanet UDS provides a horizontal scroll mechanism. On Windows NT 4.0/3.51, you can reset this property to provide multi-row tabs. If you use multi-row tabs, they will default to scroll bars on all other platforms.

### List Views and Tree Views

List views and tree views are exceptions to the rule of using widgets from the native window toolkit, because iPlanet UDS only uses native list view and tree view widgets on Windows NT. On all other platforms, iPlanet UDS uses a custom widget, the outline field. (Outline fields are defined by the OutlineField class; the ListView and TreeView classes are subclasses of—and restricted versions of—the OutlineField class.)

A complete discussion on the advantages and disadvantages of using tree or list views versus outline fields is found in [Chapter 2, “Using Complex Widgets.”](#) To summarize what to expect of list views on non-Windows platforms:

- look like outline fields, not the platform’s native list views, if any exist
- display only in the Detail style

No matter what style is specified for the list view in your application, iPlanet UDS will use an outline field display that resembles the Detail style of a list view.

- do not allow column resizing by the user
- do not allow column sorting by the user
- highlight whole rows, not separate columns

Additional information is available in [Chapter 2, “Using Complex Widgets,”](#) and in the Display Library online Help in the sections describing the ListView, TreeView, OutlineField, and DisplayNode classes.



## Fonts

For widgets that contain or display text, iPlanet UDS provides a choice between system and portable fonts.

The *system font* is the default font that a window system uses to display text on windows, including the menu bar, the fields, the data displayed by the fields, and so on. Each window system uses a different system font.

By default, all widgets in an iPlanet UDS application use the current window system's system font. Using the system font for your widgets guarantees that the application is consistent with the native window system. However, because the system font is different on each window system, the size of the text will vary from system to system. Font differences will affect alignment of widgets (especially when you are using text graphics) and may create sizing problems for widgets with labels. For example, if you create a push button with a text label that fits perfectly on the button on one window system, it may not fit the same button size on another system because the system font is larger on that system.

To ensure that data fields, text fields, and text edit fields always display the required number of characters, you can set their size using the Size Policy command. On the Size Policies properties dialog, set the field's Height and Width Policy properties to Natural to ensure that the field is always large enough to display its content. Then, set the Visible Rows and Visible Columns properties to the number of rows and columns that the field needs to display. This way, no matter what the font is, the required number of characters will always be displayed.

Note that for character fields that are in size partnerships or whose Width and/or Height Policy properties are set to Parent, you can set the minimum number of rows and columns needed for the character field. See [“Resizing Fields Within a Grid Field” on page 138](#) for information about setting a field's minimum size.

To ensure that widgets stay aligned, you should use grid fields as described under [“Using Grid Fields” on page 137](#). To ensure that widgets that display text labels, such as push buttons, are sized appropriately, use the Natural setting for Height and Width Policy properties as described under [“Field Size Policies” on page 136](#).

A *portable font* is a traditional typeface, such as Helvetica or Times Roman, that is generally available on all the different window systems. When you use a portable font, you select the type face, point size, and style. Using a portable font guarantees that the type used in the application will look similar on all window systems. However, because there are subtle variations between the same fonts on different

window systems, using a portable font does not solve the alignment and sizing problems described above. You should still use grid fields to ensure that widgets stay aligned, you should still use size policies to ensure that the widget is sized appropriately, and you should still set minimum sizes for character fields.

## Image Resolution

The size of an image that is being displayed depends on two things: the number of pixels in the image and the resolution of the monitor that is displaying the image. The number of the pixels for a given image is fixed (unless you use iPlanet UDS scaling options described in the table below). However, the resolution of a monitor can vary from workstation to workstation. The lower the resolution of the monitor, the larger the image will be. Even if all your end users are working with the same window system, you may still need to take differences in monitor resolution into account.

### *Image Size Policy for Picture Fields and Graphics*

For picture fields and picture graphics, iPlanet UDS allows you to ensure that the entire image is always displayed by scaling the image to fit the field. Or, you can ensure that the field does not change size by clipping the image to fit into the field.

The image size policy settings are the following:

Value	Definition
Natural	The image size never changes (the true image size is retained). If the field is too small for the image, the image is cropped. If the field is larger than the image, the setting of the ImageGravity attribute is used to position the image within the field.
Field	The image is scaled to fit into the field, possibly distorting the image.
Field Height	The image height is scaled to fit the field height. The width will be adjusted to preserve the aspect ratio.
Field Width	The image width is scaled to fit the field width. The height will be adjusted to preserve the aspect ratio.

Typically, it is best to use the Natural setting for an image. This not only ensures that the image looks better (it is not distorted), but it provides better performance. The scaling required for the Field, Field Height, and Field Width settings requires significant processing; therefore, you should use these settings only when necessary.

### *Picture Buttons and Palettes*

iPlanet UDS does not provide scaling for picture buttons and palette lists because these typically contain line art, which does not scale well. To ensure that images for picture buttons and palette lists are the correct size, you can provide separate images for each monitor type by creating the fields dynamically. You can use the following attributes of `task.part.WindowSystem` (described in the Display Library online Help) to get information about the current screen size and resolution:

<b>Attribute</b>	<b>Description</b>
HeightInPixels	A read-only attribute that shows the height of the screen of an application's client partition, in pixels.
HorzPixelsPerInch	A read-only attribute that shows the number of pixels per inch on the screen of an application's client partition, horizontally.
VertPixelsPerInch	A read-only attribute that shows the number of pixels per inch on the screen of an application's client partition, vertically.
WidthInPixels	A read-only attribute that shows the width of the screen of an application's client partition, in pixels.

### **Styles**

Each window system has its own conventions for the placement of elements in dialog boxes. iPlanet UDS provides two alternatives for dealing with this problem:

- dialog methods on the Window class provide portable dialog box formats, which always use the conventions of the host window system
- the WindowSystem object, which allows you to make dynamic modifications to a window based on the current window system

### *Dialog Methods*

The dialog methods on the Window class provide five options for creating dialog boxes in a portable format. The dialog box that you display using any of these methods will automatically use the conventions of the host window system. The dialog methods (described in the Display Library online Help) are:

<b>Method</b>	<b>Description</b>
FileOpenDialog	Provides a dialog that prompts the end user to select a file to be opened.

Method	Description
FileSaveDialog	Provides a dialog that prompts the end user to select the file name for saving.
MessageDialog	Provides a dialog that displays a message to the end user and waits for him to close the dialog.
PrintDialog	Provides a print setup dialog and a print job dialog.
QuestionDialog	Provides a dialog that displays a question to the end user and waits for his answer.

When you need a special dialog that cannot be created with a dialog method, for example, a property sheet, you may wish to customize the window for each window system on which it will run. You can do this dynamically by using the `WindowSystem` object for the partition. There are two ways to get the `WindowSystem` object:

- within `UserWindow` subclass methods, use the following syntax:

```
self.Window.WindowSystem
```

- within any method, use the following syntax:

```
task.Part.OperatingEnvironment
```

See the Display Library online Help for a description of the `UserWindow` class.

Note that you must cast the `OperatingEnvironment` object to a `WindowSystem` object to invoke `WindowSystem` methods on it or set its `WindowSystem` attributes.

Each window system also has different conventions for placing standard menus on the menu bar and for labeling the commands on them. The iPlanet UDS Menu Workshop provides two features to ensure that your menus are portable: prefabricated submenus and prefabricated commands.

When you include a prefabricated submenu on your menu bar, iPlanet UDS ensures that the appearance of the submenu (that is, its label and position on the menu bar) is consistent with the standard for each window system on which your application runs. For example, if you include the prefabricated Help submenu, it will appear in the correct position on the menu bar.

The three prefabricated submenus are:

Submenu	Description
File	Uses the window system's standard File menu title and position on the menu bar.
Edit	Uses the window system's standard Edit menu title and position on the menu bar.
Help	Uses the window system's standard Help menu title and position on the menu bar.

When you include a prefabricated command on a menu, iPlanet UDS ensures that the appearance of the command (that is, the text and shortcut key) and, in some cases, its behavior is consistent with the standard for each window system on which your application runs.

The prefabricated commands fall into four categories:

Categories	Description
Edit menu commands	<p>Cut, Copy, Paste, Delete, Undo, Redo, Find, Find Next, Replace, Replace Next, and Select All commands that look and behave according to your window system standards.</p> <p>The automatic behavior for the Cut, Copy, Paste, Delete, and Select All commands takes effect for subclasses of CharacterField (text field, text edit field, data field, and fillin field). The automatic behavior for the Undo and Redo commands take effect only for the text edit field.</p>
Print commands	A Print command that prints the current window and a Print Setup or Page Setup command that opens a print setup dialog. These commands are described in further detail below.
Help commands	Help Contents and Help Search commands, which automatically access the default Help file. A Help on Help command, which automatically accesses the appropriate Windows Help screen. An About command, which has no default behavior.
File menu commands	Close and Exit commands for inclusion on a File menu. These provide the window system standard name and shortcut key. However, you must implement the behavior of the command.

See the chapter on the Menu Workshop in *A Guide to the iPlanet UDS Workshops* for information about using prefabricated submenus and commands.

## Tools for Portable Displays

One result of the technique of using native widgets is that when you put a widget on a window using one window system and then run the window on a second window system, the widget will not only look different than it looked on the first window system, but it will probably not be the same size as it was on the first window system. Obviously, the size and shape differences between widgets will affect the alignment and spacing on your forms. If you do not take the necessary precautions, a form that looks well designed on one window system may end up looking sloppy on another window system.

iPlanet UDS provides three features to help you solve this problem:

- grid fields
- width and height partnerships
- width and height policies

### Grid Fields

Grid fields were designed specifically to enable you to align widgets on a form so that the form is completely portable. When you place your widgets in a grid field, the positioning of the widgets within the grid is relative, rather than fixed. For example, you could specify that the widgets in the second column of the grid field are all aligned on the left. Because the grid field ensures that the first column is always wide enough to accommodate the widest widget in the column, the widgets in the second column will always be aligned properly, no matter what window system displays them.

You can also use grid fields to automatically resize the widgets they contain. You do this by setting the width and height policies for widgets in the grid field to Parent.

For complete information about using grid fields, see [“Using Grid Fields” on page 137](#).

## Field Size Partnerships

iPlanet UDS allows you to link any number of fields together in a height or width partnership. When fields are in a size partnership, their height or width is determined by the largest minimum height or width of the fields in the partnership. A field partnership is useful, for example, for ensuring that a set of related push buttons or data fields are always the same size. In the simple dialog window shown in [Figure 3-1 on page 138](#), the two buttons are in a width partnership.

The `Arrange > Fields into Height Partnership` and `Arrange > Fields into Width Partnership` commands in the Window Workshop allow you to select a group of fields to include in a height or width partnership. The fields in the partnership can be simple or compound, resizable or fixed size.

If you include fixed-size widgets in a size partnership, the fixed-size field itself is not resized, but its minimum size affects the resizable widgets. For example, if a radio list, which has a fixed size, is in a width partnership with a push button, which is resizable, the radio list will never be resized but the push button will be sized in relation to the radio field. If all are fixed, the partnership is legal, but meaningless. All compound fields are sizeable, and can be included in size partnerships. See the section on creating simple widgets in *A Guide to the iPlanet UDS Workshops* for a list of the sizeable and fixed-sized simple fields.

When you add a field to a height partnership, its Height Policy property is automatically set to Size Partner. The height partnership is completely independent of the width partnership. Likewise, when you add a field to a width partnership, its Width Policy property is automatically set to Size Partner. The width partnership is completely independent of the height partnership.

When two or more fields are in a size partnership, iPlanet UDS uses the height or width of the field with the largest minimum height or width in the partnership to determine the height or width for all the fields in the partnership. You specify the minimum height or width by using the Size Policy command on the Widget menu.

The Minimum Height property specifies the absolute minimum height for the field. Setting a minimum height for a field ensures that iPlanet UDS will never resize the field below this minimum. Likewise, the Minimum Width property specifies the absolute minimum width for the field. For text fields, text edit fields, and data fields, you specify the minimum dimensions in rows and columns. For all other resizable fields, you specify the minimum dimensions in mils. See [“Resizing Fields Within a Grid Field” on page 138](#) for further information about minimum height and width.

Do not confuse the minimum height or width (which is usually 0 by default) of a sizeable widget with its explicit height or width (which you set when you draw the widget). At least one of the widgets in a size partnership must have a minimum height or width of non-zero, or both will shrink to a value of 1 when you set them into a size partnership.

The Window Workshop provides the following commands on the Arrange menu for setting size partnerships:

Command	Description
Fields Into Height Partnership	All selected fields are grouped into a height partnership.
Fields Into Width Partnership	All selected fields are grouped into a width partnership.
Remove Field From Height Partnership	All selected fields are removed from the height partnership.
Remove Field From Width Partnership	All selected fields are removed from the width partnership.

Note that when the fields are contained by the grid field, you can often use the Height and Width Policy settings for Parent to get the same effect as a size partnership. See [“Resizing Fields Within a Grid Field” on page 138](#) for information about the Parent setting for Height and Width Policy properties.

## Field Size Policies

The Height and Width Policy properties for fields provide a range of alternatives for adjusting the field size, allowing you to control whether the field size is static, whether iPlanet UDS adjusts the field size to accommodate its content, or whether the field size is determined by the grid that contains it.

The Natural setting for a field’s Height and Width property is especially useful for solving the problem of widget differences between various window systems. When the Height or Width Policy of a field is set to Natural, iPlanet UDS automatically adjusts the height or width of the field to accommodate the field’s content. For example if the width of a push button is Natural, the button will always be large enough to display its label.

For further information about Height and Width Policy properties, see [“Resizing Fields Within a Grid Field” on page 138](#).



## Using Grid Fields

The grid field is the most important feature that the Window Workshop provides to help you create portable user interfaces. The following sections provide information on different aspects of using grid fields.

### Nesting Grid Fields

To ensure portability, all the fields on your form should be contained by a grid field. Typically, a portable window contains nested grid fields—subsections on the form consist of grid fields that control the positioning of the individual fields within them, and the form as a whole is contained by a grid field that controls the positioning of its child grid fields.

#### *Top-level Grid Field*

A basic rule of thumb is that you should always have one grid field at the top level of the form. In other words, the immediate child of the form should be a single grid field. This ensures that all the fields within the grid field will maintain their alignment with each other.

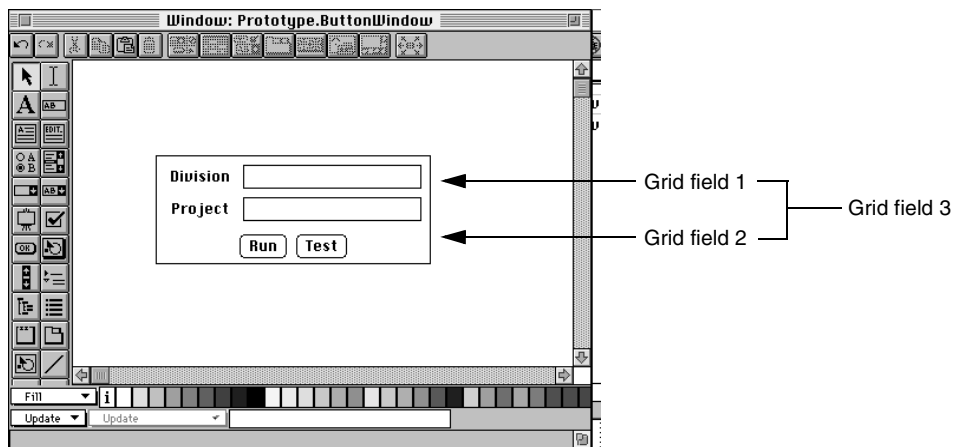
If your window is resizable, it is a good idea to set the Height and Width Size Policy settings for the top-level grid field to Parent. This ensures that when the window is resized (and the form automatically resized with it), the grid field contained by the window will automatically be resized to fit its parent. Naturally, you must also set the size policies of the individual child fields within the grid field to resize appropriately. See [“Resizing Fields Within a Grid Field” on page 138](#) for information.

#### *Child Grid Fields*

The grid fields nested within the top-level grid define relationships between the fields they contain. Generally, you should group fields into a grid field if you want them to maintain a visual relationship with each other, for example to stay aligned on the right, even if their size changes due to font or widget differences on various platforms.

A simple example is a dialog that contains two entry fields and two buttons. To keep the two entry fields aligned with each other, you would group the two data fields and their labels into one grid field. To keep the two buttons aligned with each other, you would group the two buttons into a second grid field. Finally, you would group the first and second grid fields into another grid field, the window’s top-level grid field. [Figure 3-1](#) illustrates nested grid fields:

**Figure 3-1** Nested Grid Fields



Usually, the best way to arrange the grid fields is to start by creating the innermost grid fields and finish by creating the top-level grid field. Turn on the View 7 Compound Field Lines toggle so you can see each grid field clearly outlined.

### *Aligning Labels with Data Fields*

It is not necessary to align each label in **Figure 3-1** with each data field separately. Just create a grid field around all four elements and they will align themselves horizontally and vertically. You can then change the justification of the labels (for example, if you want them all justified right instead of left) by selecting each label and using the Cell Gravity Tool to specify the alignment.

### **Resizing Fields Within a Grid Field**

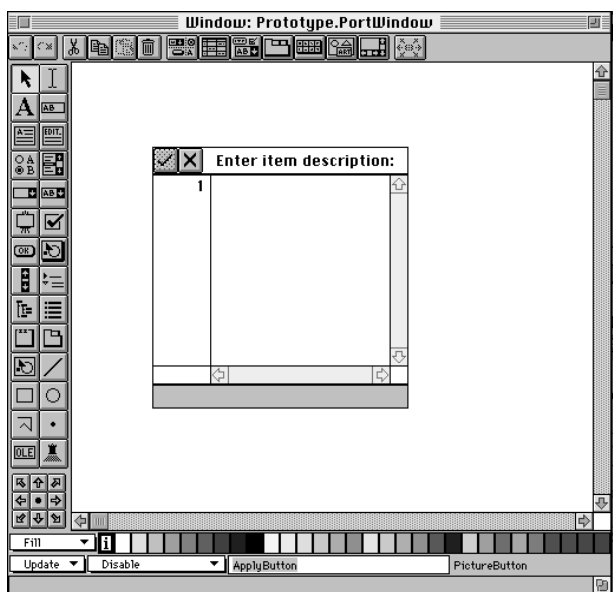
When you include a resizable field in a grid field, you can specify that its size is determined by the size of the grid field cell that contains it. This way, if the grid field expands or shrinks, either because of window system differences or because the end user explicitly resizes the grid, the fields within the grid field will expand or shrink correspondingly.

The Size Policy command Widget lets you set the field's Height and Width Policy properties. The Parent setting for the Height and Width Policy properties specifies that the height or width of the field is determined by the height or width of its parent.

If the field is in a grid field, this setting uses the grid field cell size to determine the field's height or width. When the grid field cell changes size, the height or width of the child field will automatically be resized to fit the cell. The Parent setting is available for resizable fields only; fixed-size fields cannot be resized by their grid field cells.

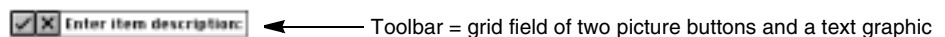
An example of the use of the Parent setting for Height and Width Policy is a simple editor window that contains a toolbar at the top, a text edit field in the center, and a status line at the bottom. The following example illustrates the completed window:

**Figure 3-2** Simple Text Editor Window



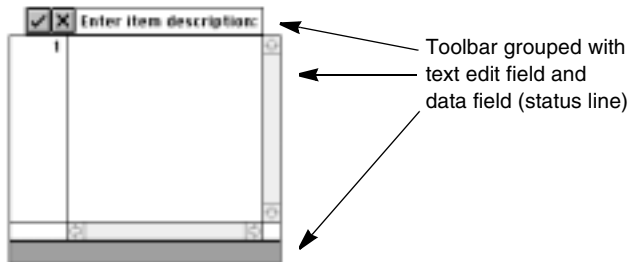
To build this example, first create the toolbar by grouping the two picture buttons and text graphic into one grid field with a frame and setting default cell gravity to Middle Left:

**Figure 3-3** First Grid Field



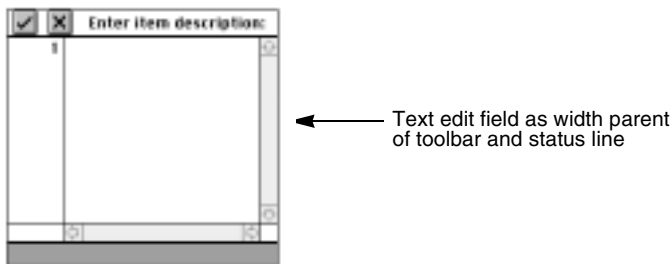
Next, create the text edit field and status line (a data field) and group them with the toolbar into the second grid field:

**Figure 3-4** Second Grid Field



Notice that the widths of the toolbar, text edit window, and status line are not coordinated. Use one of the cells as the focus for width, so when it is resized, the rest are resized accordingly. In this case, use the text edit field as focus. Set the other widgets' Width Policies to Parent. **Figure 3-5** shows the result:

**Figure 3-5** Width Policy Adjustment



## Row and Column Alignment

The Row and Column Alignment properties for a grid field specify how extra space is allocated between rows and columns in the grid field when the grid field is enlarged. If the grid field Height and Width Policy properties are not set to Natural, the grid field can be resized above its minimum. The two Alignment properties specify how the extra space is allocated.

By default, iPlanet UDS justifies the rows and columns in the grid field, adding extra space evenly to the left, right, and between the columns, and evenly above, below, and between the rows. (Note the distribution of widgets in the toolbar in **Figure 3-5**.) However, you can request that the space be allocated as follows.

For the Column Alignment property, you have the following options:

<b>Value</b>	<b>Description</b>
Left	All space is added to the right of the columns.
Right	All space is added to the left of the columns.
Center	Half the extra space is added to the left of the columns and half to the right.
Justify	The default. Space is added between and around all columns, according to column's justify weight. If the justify weight is set to 0 for all the columns (the default), the space added to the grid field is evenly distributed between all the columns.

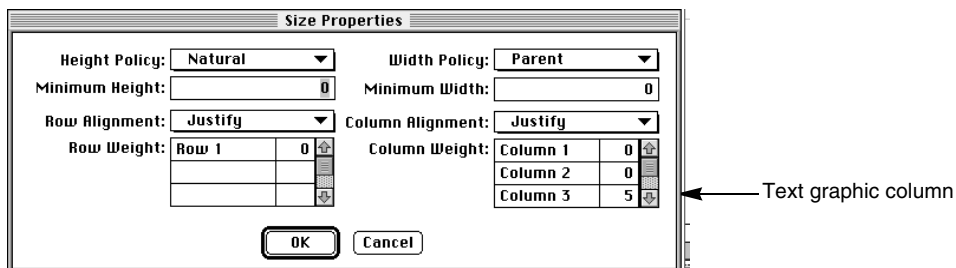
For the Row Alignment property, you have the following options:

<b>Value</b>	<b>Description</b>
Top	All space is added below the rows.
Bottom	All space is added above the rows.
Center	Half the extra space is added above the rows and half below.
Justify	The default. Space is added above, below, and between the rows, according to justify weights set for the rows. If the justify weight is set to 0 for all of the rows (the default), the space added to the grid field is evenly distributed between all the rows.

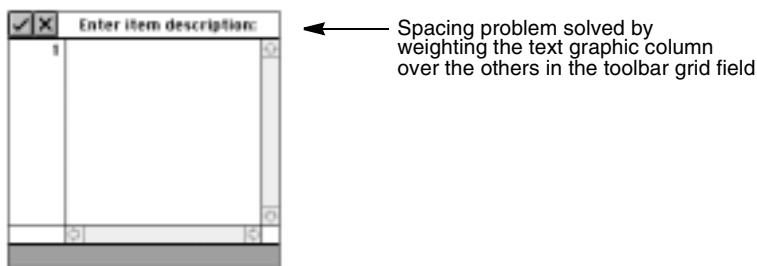
The justify weight for a grid field column or row is used to calculate the percentage of extra space allocated to the particular column or row when the grid field that contains the column or row is enlarged.

By default, the justify weight for all columns and rows is zero. When all columns or rows have a zero justify weight, any extra horizontal space is distributed evenly between the them. However, if you want certain columns or rows in the grid field to get a larger percentage of the space, you can specify the distribution explicitly.

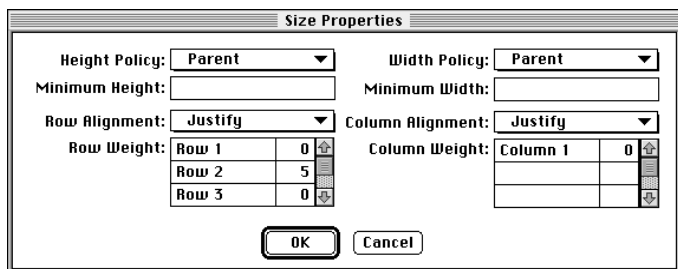
For example, as the width of the toolbar in [Figure 3-5](#) was stretched to the size of its parent (the text edit field), the spacing for its three widgets was distributed evenly, which is not the most attractive display. To make the buttons stay adjacent to each other flush left and the text graphic take up the rest of the width, set the Column Weight property for the text graphic (Column 3) to non-zero, and leave the others at 0. This gives the text graphic 100 percent of the extra space:



The result of this setting looks like this:



Also, in the grid field that includes the whole window, the text edit field is the only field that needs to expand vertically when the grid field that contains it is enlarged. To accomplish this, set the row weight property for the text edit field to non-zero, and the two other fields to 0:

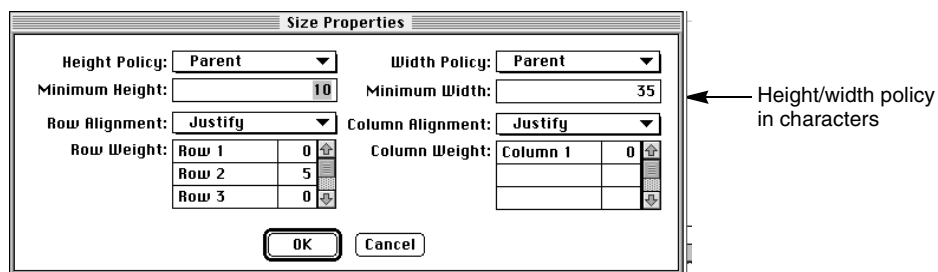


The Size Policy dialog for the grid field lets you specify the justify weights for each of the columns and rows in the grid field. You can use any numbers you like to represent percentages. The weights are converted by adding them up and calculating a percentage based on the total (normalization). If you do not set the justify weight for all of the columns or rows, the columns or rows whose justify weight you do not set keep the default justify weight of zero, and so get zero percent of the extra space.

To ensure that a field whose size is controlled by its parent never gets too small, you should specify its minimum height and width. Setting a minimum height for a field ensures that iPlanet UDS will never resize the field below this minimum. Setting minimum size also ensures that the grid field size will always be large enough to display the entire field at its specified minimum height and width. The grid field will never “clip” its children.

On the Size Properties dialog for the field, the Minimum Height property specifies the absolute minimum height for the field. For character fields, you specify the minimum height in rows and for all other fields in mils. Likewise, the Minimum Width specifies the absolute minimum width for the field. For character fields, you specify the minimum in characters and for all other fields in mils. [Figure 3-6](#) shows a possible minimum height and width for the example window.

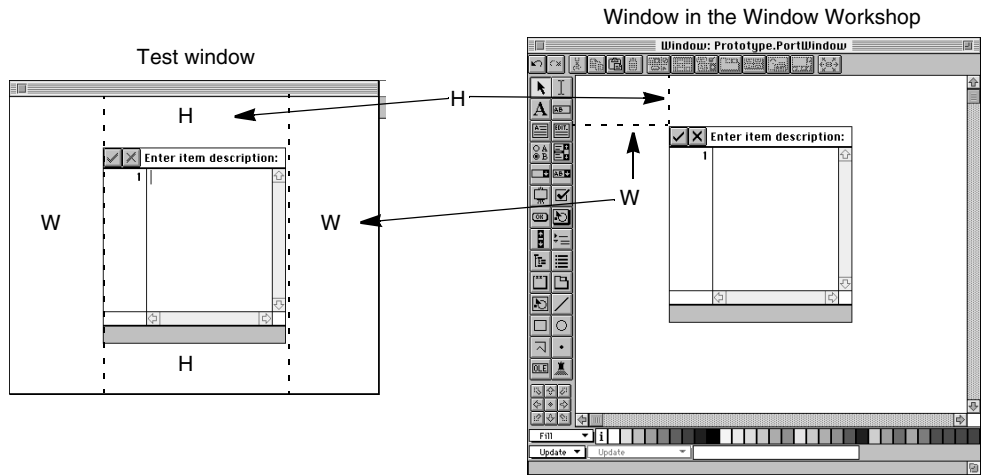
**Figure 3-6** Size Properties Dialog for the Parent Grid Field



## Specifying the Window's Border

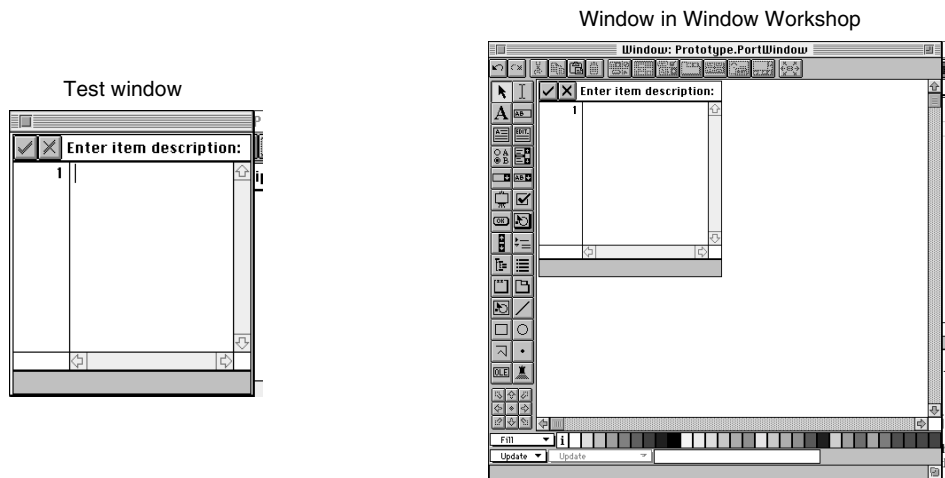
If you test the example window as it appears in [Figure 3-2 on page 139](#), you will see more space around the window than you intended. This is because the border of space around the window is determined by how much space exists in the upper left corner of the space of the window in the Window Workshop, as shown in [Figure 3-7](#):

**Figure 3-7** Testing the Window in [Figure 3-2](#)



To set the border to be flush with the window, reduce the dimensions of the corner space to zero. In other words, move the window to the upper left corner of the Window Workshop window, as shown in [Figure 3-8](#):

**Figure 3-8** Adjusting the Border of a Window





To move a grid field, position the cursor on the frame or background of the grid field and drag it to the new position. However, some grid fields do not have a background or frame, which is the case of the window in [Figure 3-8](#). In this case, create a small frame by adding margins to the grid field. You do this by adding values to the Default Cell Margin fields in the grid field's property sheet.

## Using Column and Row Partnerships

All the columns in one grid field can be in a width partnership with all the columns in another grid field. Likewise, all the rows in one grid field can be in a height partnership with all the rows in another grid field. These work the same way as the simple width and height partnerships described under [“Field Size Partnerships” on page 135](#).

Column and row partnerships are very useful when your form contains two grid fields that are separated by other widgets. When one grid field is above another, you can make their corresponding columns exactly the same width by joining the two grid fields into column partnerships. (This is the type of relationship we use in our array fields, with the column titles the same width as the body columns.) When the grid fields are side by side, you can make their corresponding rows the same height by joining the two grid fields into a row partnership.

It is important to realize that a column or row partnership only guarantees that the *sizes* of the columns or rows are the same, not their physical alignment on the form. To physically align them, you can nest both grid fields in a parent grid field. Or, for column partnerships, align both grid fields to the same left margin, and for row partnerships, align both grid fields to the same top margins.

The column and row partnership commands are:

Command	Description
GridFields Into Row Partnership	Links selected grid fields into a row partnership.
GridFields Into Column Partnership	Links selected grid fields into a column partnership.
Remove GridField From Row Partnership	Removes selected grid field from a row partnership with other grid fields.
Remove GridField From Column Partnership	Removes selected grid field from a column partnership with other grid fields.



# Implementing Online Help

iPlanet UDS provides support for implementing an online help system, as well as float-over and status-line help.

This chapter provides information about the following topics:

- creating float-over and status line help for fields
- creating status-line help for menu items
- building a context-sensitive online help system
- using pre-fabricated help commands

## Overview

iPlanet UDS supports two kinds of online help:

- terse help text that is displayed in status lines and in float-over tips
- screens of text that users can access from within an iPlanet UDS application, but are created and stored outside of the application

You create and store float-over and status-line help text within your iPlanet UDS application. You can develop your online help source text using any of a wide variety of third-party tools.

iPlanet UDS provides several features that support the development of online help. Most of your online help requirements can be met using features built in to the iPlanet UDS Workshops. However, iPlanet UDS also provides a programmatic interface to the Windows Help API if you wish to use any of the special features that Windows help provides, such as positioning the help window, or raising the “Search” window.

The following is a brief summary of iPlanet UDS's support for implementing online help:

- `Widget > Help Text` command in the Window Workshop

Allows you to assign a topic in the default help file to a field. iPlanet UDS displays this help topic when the input focus is on the field and the end user presses the Help key. See [“Context-Sensitive Help” on page 149](#) for information.

Also allows you to specify both float-over and status-line help for the field. Float-over help is displayed alongside the field when the mouse pauses over the field. Status-line help is displayed in the window's status line when the mouse pauses over the field.

- `Window.StatusText` attribute

Used in conjunction with a status-line widget, allows you to create status-line help text when the cursor pauses over a widget or menu item.

- Prefabricated Help menu

The Menu Workshop provides three prefabricated commands for inclusion on the Help menu: Help Contents, Help Search, and Help on Help. The Help Contents and Help Search commands open the appropriate Help dialog, using the default help file you have specified for the window, or if none exists for the window, for the partition. The Help on Help command displays Windows standard Help on Help window.

You can override the automatic behavior of the three iPlanet UDS Help commands by providing your own TOOL code to handle the Activate events on these commands.

iPlanet UDS also provides an About command, for which you can provide the appropriate processing. See [“Using the Prefabricated Help Commands” on page 160](#) for information about implementing a Help menu for your window.

- Methods and events

The `WinHelp` method defined on the `WindowSystem` class provides an interface to the Windows Help API. You can use this method to access any Windows Help documents. See `WinHelp` method on `Window` class in the Display Library online Help for further information.

The `HelpRequest` event defined on the `Window` class is posted when the end user presses the Help key. You can provide any processing you wish in response to this event. See `HelpRequest` event on `Window` class in the Display Library online Help for further information.

## Context-Sensitive Help

Context-sensitive help is a help screen of information relevant to the field that has input focus when the user presses the Help key. You create context-sensitive help by first developing a help file that contains a set of help topics. Each topic is uniquely identified and associated with a piece of your application.

When the end user presses the Help key, iPlanet UDS checks the field that has the input focus to see if a value was provided for its help topic (stored in its `HelpTopic` attribute). If there is a help topic associated with the field that has the input focus, iPlanet UDS displays the field's help topic. If there is no help topic associated with the field that has the input focus, iPlanet UDS checks the field's parent, the parent's parent, and so on, all the way up the containment hierarchy to the window, until it finds a help topic. If there is no help topic associated with the window, iPlanet UDS displays the Contents page for the default help file.

Only certain fields can have the input focus—these fields differ between window systems. For example, all window systems give the input focus to text fields. However, only some window systems give the input focus to push buttons. If you are creating context-sensitive help that uses input focus events, you need to be aware of the differences between the window systems. See the Display Library online Help for more information about input focus.

### Default Help File

The default help file is any Windows help document in the appropriate format. You can provide an individual help file for each window in your application or one file for the application as a whole. When you are implementing context-sensitive help, the help file should include a topic for each field that needs one and a topic ID number, and, if desired, a key word associated with each topic.

### DefaultHelpFile Attribute

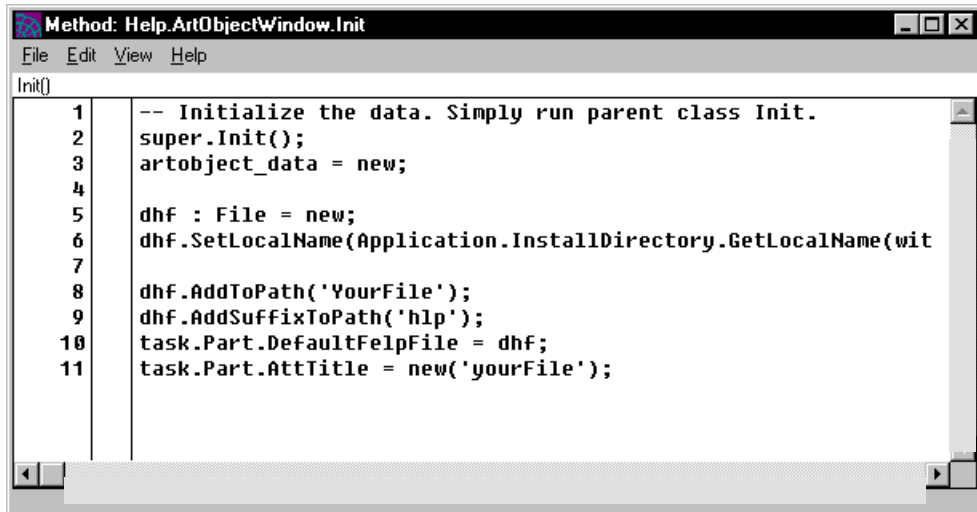
You must specify the default help file for the window or partition in your TOOL code. The `DefaultHelpFile` attribute of the Window object specifies the default help file for an individual window, and the `DefaultHelpFile` attribute of the Partition object provides the default help file for the application as a whole. See the Display Library online Help for information about the `DefaultHelpFile` attribute. If there is no default help file, context-sensitive help for the field will not be displayed.

► **To specify the DefaultHelpFile attribute**

1. Create a Windows Help document in the appropriate format.
2. Set the DefaultHelpFile attribute to the name of the help file.

Remember that you can set the DefaultHelpFile attribute for the Partition object to specify a default help file for the entire application, or set the attribute for each of the Window objects to specify a default help file for each window.

In the example that follows, the default help file is specified using portable naming conventions, so that the file is portable across platforms. See the Framework Library online Help for information about the SetLocalName attribute.

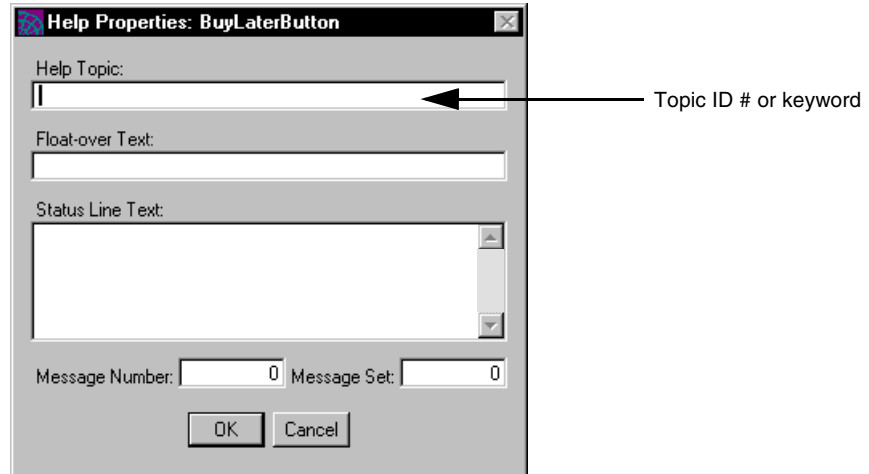


```
Method: Help.ArtObjectWindow.Init
File Edit View Help
Init()
1  -- Initialize the data. Simply run parent class Init.
2  super.Init();
3  artobject_data = new;
4
5  dhf : File = new;
6  dhf.SetLocalName(Application.InstallDirectory.GetLocalName(wit
7
8  dhf.AddToPath('YourFile');
9  dhf.AddSuffixToPath('hlp');
10 task.Part.DefaultFelpFile = dhf;
11 task.Part.AttTitle = new('yourFile');
```

► **To provide context-sensitive help**

1. Enter the Window Workshop.
2. Select the widget for which to provide help and choose the Widget > Help Text command.

The Help Properties dialog appears:



3. Specify the topic ID number or help key word for the topic in the help file you wish to display and click OK.
4. Repeat **Step 2** and **Step 3** for each widget that has an associated help topic.

The help topic you assign to a widget can either be the ID number of a help topic in the default help file or, if you have set up key words for the help topics, a key word that is associated with one or more topic ID numbers in the default help file. If you specify a topic ID number, iPlanet UDS displays the help message associated with the ID number. If you specify a key word, iPlanet UDS displays the help message for the *first* topic ID associated with the key word. The following table shows the syntax for both help topic assignments:

Help Topic Specification	Description
<i>#topic_ID</i>	A pound sign followed by a topic ID number specifies the identification number of a help topic in the default help file.
<i>string</i>	A string specifies a previously defined help key word that is associated with a topic ID number in the default help file. If the key word is associated with more than one help topic ID, iPlanet UDS uses the first topic.

## Float-Over Help

Float-over help is a help message that is displayed next to the field whenever the mouse pauses over the field. To provide float-over help for the fields on your window, you must ensure that float-over help is turned on for the window system, and you must provide the float-over help text for the individual fields on the window.

### Enabling Float-Over Help

The `IsFloatOverEnabled` attribute of the `WindowSystem` class specifies whether or not the float-over help for the window system is displayed. The default value of the `IsFloatOverEnabled` attribute is determined by the `FORTE_ISFLOATOVERENABLED` environment variable, whose default value is `TRUE`. You should use the `IsFloatOverEnabled` attribute in your `TOOL` code to ensure that float-over help is turned on when appropriate, and to give the end user the option of turning it off and on. See the Display Library online Help for information on the `IsFloatOverEnabled` attribute.



## Providing Float-Over Help Text

The float-over text for an individual widget is stored in the `FloatOverText` attribute defined on the `FieldWidget` class. See the Display Library online Help for information. You enter the actual help text in the Window Workshop using the `Widget > Help Text` command.

### ► To provide float-over help for a field

1. Enable float-over help in your TOOL code by using the `WindowSystem.IsFloatOverEnabled` attribute and setting its value to `TRUE`.

The default value is derived from the `FORTE_ISFLOATOVERENABLED` environment variable, whose default value is `TRUE`.

2. In the Window Workshop, select the widget you wish to provide float-over help for and choose the `Widget > Help Text` command.

The Help Properties dialog appears.



3. Enter the float-over help message in the Float-Over Text field.
4. Repeat steps 2 and 3 for each widget that will use float-over help.



## Float-Over Help for Palette Lists

To provide float-over help for individual regions (or icons) in a palette, you must use the Float-Over Text property on the palette's Properties dialog for each list item in the palette list. See *A Guide to the iPlanet UDS Workshops* for information about creating palette lists.

► **To provide float-over help for a palette list**

1. Create your palette list field.
2. Double-click on the palette list, or select the palette list and choose the Widget > Properties command.

The PaletteList Properties dialog appears.

Value	Image	Float-Over Text	Status Line Text	Msg Number
1		Assign rules		0
2		App Dictionary		0

3. Enter the float-over help text for each item in the Float-Over Text array field.
4. Click OK.

When you pause the cursor over a list element, the corresponding text you entered in the Float-Over Text field is displayed.

## Suppressing Float-Over Help Text

If you wish to disable the float-over help text for the individual list elements, you can do so using the `ShowRegionFloatOver` attribute on the `PaletteList` class. By default, this property is set to on. You may want to set this property to off if you are using the Float-Over Help property of the palette list for your own use and do not want to display the text to the end user.

See the Display Library online Help for information about the `PaletteList.ShowRegionFloatOver` attribute.

## Status-Line Help

Status-line help is help text that is displayed in the window's status line when the mouse pauses over the field.

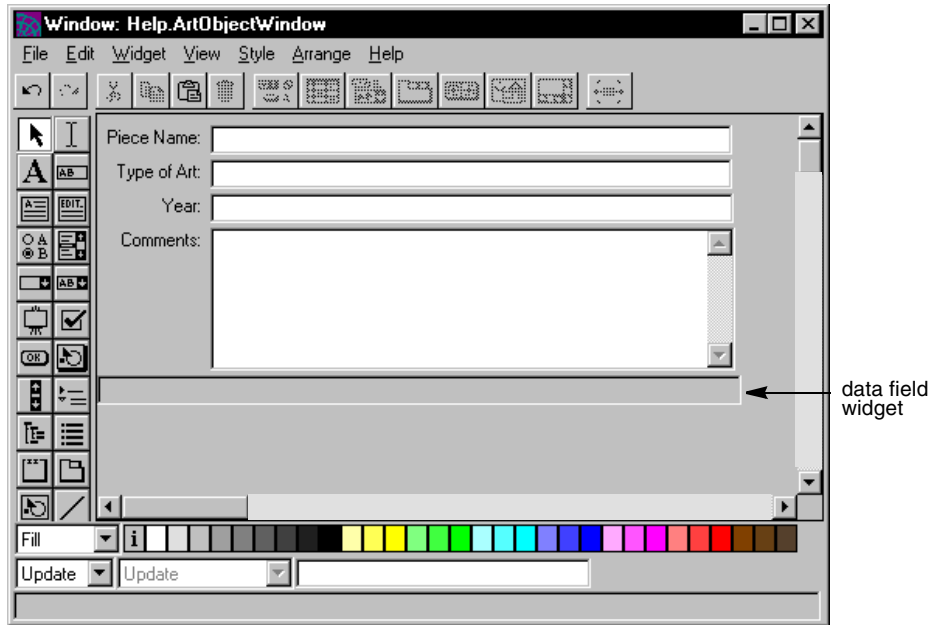
To provide status-line help for a window, you must do the following:

- use a data field widget to add a status line to the window and map this field to a `TextData` attribute in the widget's Properties dialog
- map the `TextData` attribute associated with the status line field to the `StatusText` attribute of the Window object
- provide the status-line help text for individual fields by using the Help Text command on the Widget menu

Once you have set up your status line field, each time a user moves the mouse onto a new field, iPlanet UDS automatically sets the value of the `WindowSystem.StatusText` attribute to the status-line help text value that was specified for the current field. The window is automatically refreshed, and the new value of the `StatusText` attribute is displayed in the window's status line field. If the `Window.StatusText` attribute is `NIL`, there will be no status-line help for the window. See the Display Library online Help for information about the `StatusText` attribute on the `WindowSystem` class. See *A Guide to the iPlanet UDS Workshops* for information about creating a status line field.

► **To create a status line widget**

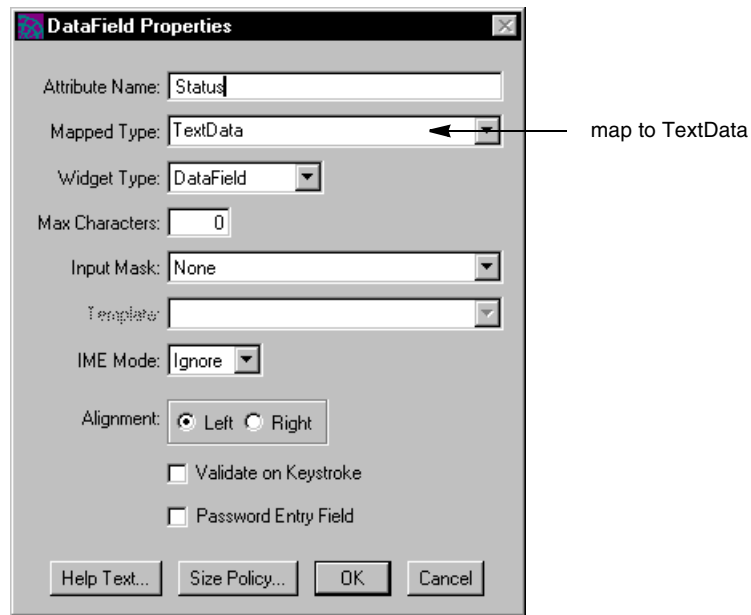
1. Create a data field widget on window where you wish to display status-line help.



2. Double-click the data field, or select the data field and choose the Widget > Properties command.

The DataField Properties dialog appears

3. Map the data field widget to a TextData attribute.



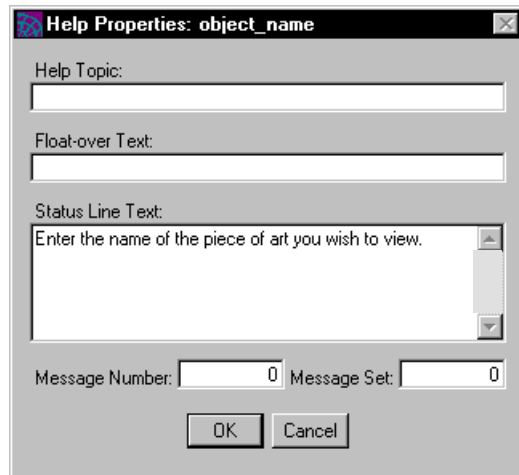
4. In your code, set the value of `Window.StatusText` to the TextData attribute that is mapped to your data field.

```

Method: Help.TestHelp.Display
File Edit View Help
Display()
1  status = new();
2  self.Window.StatusText = status;
3
4  self.Open();
5  event loop
6      when task.Shutdown do
7          exit;
8      end event;
9  self.Close();

```

5. In the Window Workshop, for each field that needs status-line help, select the widget and choose the Widget > Help Text command. In the Status-Line Text field on the Help Properties dialog, enter the status-line help message.



Note that the status-line text for an individual widget is stored with its float-over help in the FloatOverText attribute defined on the FieldWidget class. See the Display Library online Help for information.

### Status-Line Help for Palette Lists

To provide status-line help for individual regions (or icons) in the palette, you must use the Status Line property on the Palette List's properties dialog for each picture graphic or picture button in the palette.

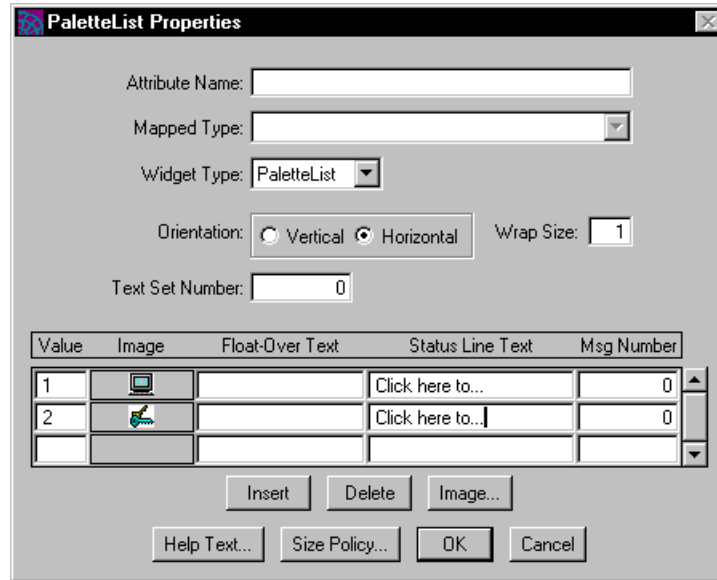
#### ► To create status-line help for palette lists

1. Create a status line widget on your window and map it to the StatusText attribute of the Window object, as described in ["Status-Line Help" on page 155](#).
2. In the Window Workshop, create a palette list.

See *A Guide to the iPlanet UDS Workshops* for information about creating palette lists.

3. Double-click on the palette list, or select the palette list and choose the Widget > Properties command.

The PaletteList Properties dialog appears.



4. Enter status-line text for each list item and then click OK.

## Status-Line Help for Menu Widgets

You can provide status-line help for individual menu items that is displayed on the Windows and Motif window systems. On all other platforms, the status text associated with a menu item is ignored. Status-line help for a menu item is displayed when the mouse pauses over the menu item and status-line help for the window is turned on.

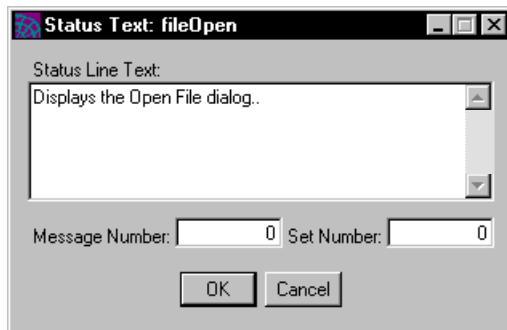
### ► To provide status-line help

1. Create a status line widget on your window and map it to the StatusText attribute of the Window object, as described in [“Status-Line Help” on page 155](#).
2. In the Menu Workshop, create the menu for your window.

For information about creating menus, see *A Guide to the iPlanet UDS Workshops*.

3. Select a menu command for which you wish to provide status-line text and choose Item > Status Text.

The Status Text dialog appears.



4. Enter the status-line text in the Status Line Text field and click OK.
5. Repeat steps 3 and 4 for each menu command.

Each time the mouse moves onto a menu item, iPlanet UDS automatically sets the value of the `WindowSystem.StatusText` attribute to the status text value you specified for the menu item with the Item > Status Text command in the Menu Workshop. The window is automatically refreshed, and the new value of the `StatusText` attribute is displayed in your window's status line field.

If `Window.StatusText` is NIL, there is no status line help displayed for the window.

## Using the Prefabricated Help Commands

The Menu Workshop provides a prefabricated Help menu and four prefabricated Help commands. iPlanet UDS uses the title and position on the menu bar that is customary on each particular window system, so using the prefabricated Help menu ensures that your Help menu is portable.

The iPlanet UDS Help commands that you can include on the Help menu are the following:

Help Command	Description
Help Contents	Displays the Help Contents window of the default help file.
Help Search	Displays the Help Search window of the default help file.



Help Command	Description
Help on Help	Displays the standard Help on Help window provided by Windows.
About	Provides an About command in the appropriate location. You must register for the AboutMenuActivate event and provide processing to display the appropriate window.

The iPlanet UDS prefabricated Help menu commands get their help topics from the default help file. Therefore, you should be sure to include the information needed for both the Help commands and for context-sensitive help in your Help document.

Of course, if you wish to provide your own help facilities, you can override the default behavior of any of these commands by registering for their Activate events. When you explicitly handle the Activate event on the Help Contents, Help Search, and Help on Help commands, iPlanet UDS's automatic help facilities are ignored. If you wish to interact directly with Windows Help, you can use the iPlanet UDS WinHelp method defined on the WindowSystem class. See the Display Library online Help for information.

► **To include a Help menu on a window**

1. Create one Windows Help document for the application, or one for each window in the application.

The Help document should include the information needed by the Help Contents and Help Search commands.

2. Add the Help submenu to the menu bar for each window where you wish to provide help.

iPlanet UDS positions the Help menu in the appropriate place for the window system, and provides the correct title.

3. Add the Help Contents, Help Search, and Help on Help prefabricated commands to the Help menu.

You can also include the About command, but must take extra steps after doing so (described under **“Implementing the About Command” on page 162**).

4. Set the DefaultHelpFile attribute for the Partition or for each of the Window objects to the help file you wish to use.

## Default Help File

To use the Help Contents and Help Search commands, you must provide a default help file, either for the application as a whole or for each of the individual windows in the application that include the Help commands. (The Help on Help command uses the appropriate help screen that Windows provides.) See [“Default Help File” on page 149](#) and [“DefaultHelpFile Attribute” on page 149](#).

If there is no default help file, the Help Contents and Help Search commands cannot provide access to Help.

## Implementing the About Command

iPlanet UDS provides a prefabricated About command. Most applications use an About command to display information that describes the application. iPlanet UDS does not provide automatic behavior for the About command, but if you use the command on your Help menu, iPlanet UDS ensures that it will be portable across platforms.

To allow you to implement the behavior of the About command, iPlanet UDS provides a special AboutMenuActivate event, which allows your application to detect when the end user chooses the About command. You must register for the AboutMenuActivate event and provide processing to open the appropriate window.

The label for the About command consists of two parts: the word “About” and the title of your application. You specify the application title by using the AppTitle attribute on the Partition object.

### ► To use the About command

1. Include the About command on the Help menu.
2. Set the AppTitle attribute on the Partition object to the application title you wish to have displayed in the About command menu item.
3. Create an About window for the application.
4. Register for the AboutMenuActivate event on the About menu item and provide processing to display the About window you created in the previous step.

# Testing the User Interface

This chapter provides information about testing iPlanet UDS applications that are in development. It describes how to use the following utilities to test an application's user interface:

- the TestClient utility, which allows you to start one or more clients to simulate client activity
- the AutoTester project, which allows you to capture user input from a test session that you can replay while capturing state information

The AutoTester project is included on the iPlanet UDS installation media and is described in [Appendix A on page 629](#).

## Using the TestClient Utility

The iPlanet UDS TestClient utility allows you to test a shared service object by running the application on multiple clients. The TestClient utility enables a client node to participate in the testing of a distributed application that is currently being tested from a Partition Workshop. After you start the TestClient utility on a client node in the environment, you can select any application currently being partitioned in the environment to test.

The TestClient utility can test only one application at a time. However, you can start the TestClient utility multiple times on a client to test different applications simultaneously.

## Starting the TestClient Utility

To start the TestClient utility, use the **tclient** command or the tclient icon on Windows. Use the **tclient** command flags to identify the node name or model node name, the space to use for the memory manager, and any logger flags.

### Portable Syntax

```
tclient [-fnd node_name] [-fmn model_node_name] [-fm memory_flags] [-fst integer]
        [-fl logger_flags] [-fns name_server_address] [-fterm] [-fcons]
```

### OpenVMS Syntax

#### VFORTE TCLIENT

```
[/NODE=node_name]
[/MODEL_NODE=]
[/MEMORY=memory_flags]
[/STACK=integer]
[/LOGGER=logger_flags]
[/NAMESERVER=name_server_address]
[/FTERM]
[/FCONS]
```

Flag	Description
-fnd <i>node_name</i> /NODE= <i>node_name</i>	Specifies the node name to use for the session. If you do not specify one, the default depends on the operating system. On Windows and Mac, the default is set by the FORTE_NODENAME environment variable. On all other platforms, the actual node name is used.
-fmn <i>model_node_name</i> /MODEL_NODE= <i>model_node_name</i>	Specifies the model node name to use for the session. If you do not specify one, iPlanet UDS uses the value of the FORTE_MODELNODE environment variable. If the environment variable is not set, the node is not treated as a model node.
-fm <i>memory_flags</i> /MEMORY= <i>memory_flags</i>	Specifies the space to use for the memory manager.
-fst <i>integer</i> /STACK= <i>integer</i>	Specifies the thread stack size in bytes for iPlanet UDS and POSIX threads. This specification overrides default stack size allocation. For more information, refer to the <i>iPlanet UDS System Management Guide</i> .

Flag	Description
-fl <i>logger_flags</i> /LOGGER= <i>logger_flags</i>	Specifies the logger flags to use for the session.
-fns <i>name_server_address</i> /NAMESERVER= <i>name_server_address</i>	Specifies the name server to use for the session.
-fterm /FTERM	(Clients only) Specifies that the client session run as always attached to the terminal so that it always responds to terminal commands such as Control-c.
-fcons /FCONS	Displays the trace window. By default, the trace window is iconized on Windows. Use this flag to display the trace window on startup.

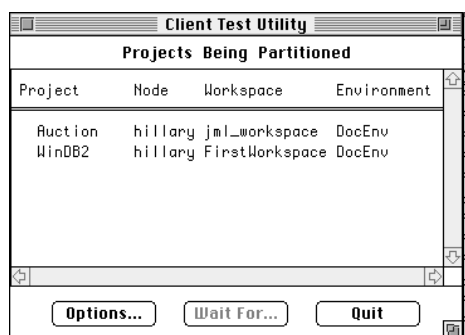
When you start TestClient, the TestClient Window appears.

## The TestClient Window

The TestClient Window displays a list of the applications currently being partitioned by developers in the Partition Workshop. This list is automatically updated every 5 seconds. You can change the refresh interval using the Options dialog (see [“Setting TestClient Options” on page 166](#)).

Figure 5-1 illustrates the TestClient Window.

**Figure 5-1** TestClient Window



## Setting TestClient Options

In the TestClient Window, the list of applications being partitioned is automatically refreshed every 5 seconds. If you wish to change the refresh interval, click the **Options...** button. In the Client Test Options dialog, enter the interval in the Refresh Interval field. The refresh interval can be any positive integer representing a number of seconds.

## Leaving the TestClient Utility

When you finish testing the application, you can exit the TestClient utility by clicking the Quit button in the TestClient window. However, as long as you are testing an application, the application blocks the TestClient window. So you must first click the Cancel button in the Wait dialog to end the test for the client; then you can then exit the TestClient utility.

## Testing the Client

The TestClient window displays a list of all the applications currently being partitioned in Partition Workshop in the current environment. The client that is running the TestClient utility can participate in testing any of these applications.

To test the application from the client, you select the application you wish to test. You then click the Wait For... button to notify the Partition Workshop that you are ready to participate in the test.

### ► To test an application

1. In the TestClient window, select the application you wish to test.
2. Click the Wait For... button.

The Wait dialog opens, which provides the Cancel button for cancelling the test if necessary.

3. When the developer uses the Run command for the selected application from the Partition Workshop, the client partition for the application will start up on your client.
4. Test the application from the client until the application exits or until the developer running the Partition Workshop uses the Stop Remote Partitions... command, which shuts down TestClient and the Wait dialog.

While the distributed application is running, the user interface displays on your client. You can exit the test at any time by clicking the Cancel button on the Wait dialog.

As long as the application continues to run on the client, you can interact with it to simulate end user activity. When the test is over, TestClient is still running and you can participate in another test if you wish.

## Using the AutoTester Project

The iPlanet UDS AutoTester project helps automate the testing of iPlanet UDS GUI applications. The AutoTester project allows you to capture user input, replay captured input, and dump state information on widgets. You can capture output from selected runs of the test project as “canonical” output—that is, output against which you can compare the results of subsequent runs. For example, you can dump a particular widget’s state after several actions to verify that its current value is what you expect, based on the canonical output.

The AutoTester project is provided with the iPlanet UDS example programs, in .pex file format. You can modify the AutoTester project for your specific testing needs. For more information on installing AutoTester, see [Appendix A, “iPlanet UDS Example Applications” on page 629](#).

Also see the Capture and Playback classes in the Display Library online Help for more information about methods and attributes used in AutoTester.

The remainder of this chapter describes how to create a suite of tests for an iPlanet UDS application using the AutoTester project. It describes how to prepare for the process, capture input, replay captured input, analyze the results, and create automated regression test suites.

## Capturing Input in an Input Capture Class

The AutoTester project is used to capture (save) input from one or more test runs of an application in development. These test runs can then be replayed and used for comparison, on different platforms, and with optional state information.

AutoTester captures and replays input as TOOL code. Specifically, the input capture process records every action taken by a user during one unique test, expressed as one or more method invocations against the iPlanet UDS Display system.

The actual output of the input capture process is an *input capture class*, which is exported as a class (.cex) file. Each input capture class contains two methods that you see when you later import the file:

- the InputDriver method, which contains all the user actions on the Display system
- the Runit method, which replays those actions against the test target application

This .cex file is suitable for direct inclusion into a test driver application.

Because AutoTester expresses all captured information in mills relative to a FieldWidget object, the captured input is portable and can be used on different platforms. You can add code to verify the application's state at key points. A test can directly access the application itself during playback; thus you can check internal features for correctness at the appropriate points.

iPlanet UDS creates *Session ID* strings to uniquely identify a field at runtime. During input capture, every window is assigned a unique sequence number at open time. Similarly, at playback time, a unique sequence number is assigned to each window as it is reopened in the same sequence. Inside each window, every field can be uniquely identified by its heritage in the containment hierarchy. For example, a field might be the seventh child of the third child of the fourth window. A name is added, if the field has one:

```
"CancelButton{4:1.3.7}Cancel "
```

When the input capture class is written, all fields are identified by a Session ID instead of an actual object address, because there is no way to remember that address until playback time.

At playback, the field name component of the Session ID is used to identify the field, if possible. The numeric part of the Session ID is used only if the field name is not adequate to uniquely identify the field.



## Setting Up for Input Capture

To create a test for an iPlanet UDS application, you need the following projects:

Project	Description
Target test project	This is the project you wish to test.
AutoTester project	This project is the engine of the test suite creation process. AutoTester is provided with the iPlanet UDS example programs.
TestDriver project	The TestDriver project drives the target test project in capture and playback mode. You create this project and include the other two projects as supplier plans (the target test project and the AutoTester project).

To illustrate how to use AutoTester, we will use the iPlanet UDS example program PencilPlay as the target test project.

### ► To set up for input capture

1. Start up iPlanet UDS.
2. Make sure your workspace contains the AutoTester project and the iPlanet UDS example program PencilPlay.

If one or both of these projects is not in your workspace, you can import them from the iPlanet UDS example program directory. Select > Plan/Import... and choose autotest.pex and/or pencil.pex from the directory named \$FORTE\_ROOT/install/examples/display.

3. Create a project named PencilPlayTester, by selecting the Plan > New Project command.

In the Project Properties dialog, turn off the Database toggle (leave the Display toggle on).

4. In the PencilPlayTester project, include PencilPlay and AutoTester as supplier projects, by selecting the File > Supplier Plans... command.
5. Copy the MyDriver class from the AutoTester project into the PencilPlayTester project.

The easiest way to do this is to open both the PencilPlayTester and AutoTester Project Workshops, then drag and drop the MyDriver class from AutoTester to PencilPlayTester.

6. Select the File > Start Class Method command, and set the start class of the PencilPlayTester to MyDriver. Then set the starting method to Runit.
7. Open the MyDriver class and edit the RunApp method to look like the following (you must uncomment the last two lines):

```
-- Create an instance of the Target App's starting class
-- and invoke its starting method.
--
-- a : TestProjectName.StartingClass = new;
-- a.StartingMethod();
--
a : PencilPlay.Window1 = new;
a.Display();
```

8. Compile and save the PencilPlayTester project.

Now you can run PencilPlay and capture input.

## Capturing Input

To run your application in input capture mode, you run the TestDriver project.

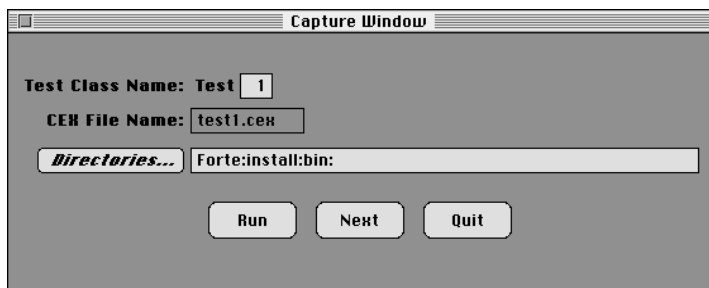
To test the PencilPlay project and capture input, you run the PencilPlayTester project. The following window appears:

**Figure 5-2** AutoTester Control Panel



This window allows you to select Capture or Replay modes.

If you click on the Capture button, the Capture Window appears:

**Figure 5-3** Capture Window

This window allows you to specify the name and location of the .cex file for the input capture class. This window has the following fields:

Field	Description
Test Class Name	Specifies the name of the class to hold the test run you are about to create. When you first enter the Capture Window, this field is set to its default value: Test 1. You can change the value for this field to any value from 1 to 99.
CEX File Name	Specifies the name of the .cex file to contain the new test class. The test class will be exported to the .cex file automatically. When you first enter the Capture Window, this field is set to test1.cex.
Directories	Designates the directory to hold the exported .cex file. This field defaults to the current directory. You can change the value for this field directly or you can click on the Directories button to choose from available directories.

► **To capture input**

1. In the Capture Window, click on the Run button to start input capture.

The target test project (PencilPlay) application starts.

2. Simulate activity in the target test project.

In the PencilPlay application, create some geometric forms, move them around, and delete them. Your input is recorded in the designated “.cex” file.

3. Exit PencilPlay by closing the window.

You return to the Capture Window. You can run another test or quit.

To run another test, create a new input capture class file and repeat these steps. If you click on the Next button, both the Test Class Name value and corresponding CEX File Name value increment by 1. Or, you can enter any value between 1 and 99 in the Test Class Name field, and the CEX File Name field value changes accordingly.

4. Quit the Capture Window.

## Dumping State Information

You can dump state information and perform other runtime validations using AutoTester.

During input capture, you can use special key sequences to insert code into the input capture class file (.cex file). The code invokes the DumpState method, described below. Then, during playback, the DumpState method invokes the DumpWidget method, which writes state information for the current widget to the dump file (.out file). Thus, you can obtain state information of varying types during each playback of captured input.

The following table shows the key sequences to use on each client platform:

Client Window System	Key Sequence
Windows	Ctrl-Shift-0 to Ctrl-Shift-9
Motif	Ctrl-0 to Ctrl-9

For example, pressing Ctrl-Shift-5 on a PC might insert the following line in a .cex file:

```
self.Dumpstate(type=5, mouseOver='{1:1.1.2.3}',
               windowSystem = ws);
```

The `DumpState` method is not defined for a predefined `iPlanet UDS` class. Instead, the class `GenericDriver` in the `AutoTester` project defines a sample `DumpState` method, with the following parameters:

The **type** parameter indicates which key sequence was used.

The **mouseOver** parameter encodes the `SessionID` of the field the mouse was floating over at the time the key sequence was used.

The **windowSystem** parameter specifies the window system on which to play back the task.

The `DumpState` method as defined by the `GenericDriver` class follows:

```
pb : Playback = ws.Playback;
pb.PlaybackTask = task;

case Type is
  when 0 do
    pb.DumpWidget(mouseOver, TW_STATE);
  when 1 do
    task.part.logmgr.put('CTRL-1 not implemented');
  when 2 do
    pb.DumpWidget(mouseOver, TW_GEOMETRY);
  when 3 do
    pb.DumpWidget(mouseOver, TW_VISUALS);
  when 4 do
    pb.DumpWidget(mouseOver, TW_VALUE);
  when 5 do
    pb.DumpWidget(mouseOver, TW_TEXT_VISUALS);
  when 6 do
    pb.DumpWidget(mouseOver, TW_ALL);
  when 7 do
    task.part.logmgr.put('CTRL-7 not implemented');
  when 8 do
    task.part.logmgr.put('CTRL-8 not implemented');
  when 9 do
    task.part.logmgr.put('CTRL-9 not implemented');
end case;
```

You can also write your own `DumpState` methods for other custom validation purposes.

This DumpState method calls the DumpWidget method defined on the Playback class in the Display library. The DumpWidget method takes two parameters: widget and outputMask. Depending on which outputMask is passed, DumpWidget provides the following diagnostic data on the widget and all its descendants:

Value for outputMask	Definition
TW_STATE	State information.
TW_GEOMETRY	Information on size and position.
TW_VISUALS	Information on visual attributes (for example, color).
TW_VALUE	The widget's current value, if any.
TW_TEXT_VISUALS	Visual Text attributes (for example, PushButton label).
TW_ALL	All the above information.

TW\_VALUE and TW\_STATE are both good choices for producing canonical output that will be valid across multiple platforms. Use the special key sequences ending in 0 and 4 to get AutoTester's Dumpstate method to call DumpWidget with the TW\_STATE and TW\_VALUE masks.

## Making Portable File References

If you want to make portable references to external files when you capture and playback input, you can set the FORTE\_AUTOTESTER\_ROOT environment variable. Note that when you use the FileSaveDialog or FileOpenDialog methods on the Window class, the captured file names are relative to the path specified by FORTE\_AUTOTESTER\_ROOT, if possible.

When you do not set FORTE\_AUTOTESTER\_ROOT, the capture and playback mechanisms assume the file names set in FileSaveDialog and FileOpenDialog are in a local, fully qualified form. It is unlikely that these names will be portable across platforms.

However, if you set `FORTE_AUTOTESTER_ROOT` during capture, the file names are captured in a portable form. You must set `FORTE_AUTOTESTER_ROOT` to a portable description of a directory. For example:

```
setenv FORTE_AUTOTESTER_ROOT /forte1/d/test/stuff
```

When you run in capture mode, you could select the following file from either of the file selection dialogs:

```
/forte1/d/test/stuff/pictures/logo.bmp
```

This file will be captured as the following portable file name:

```
%{FORTE_AUTOTESTER_ROOT}/pictures/logo.bmp
```

At playback time, this file name is rendered relative to the current setting of `FORTE_AUTOTESTER_ROOT`. This allows you to build tests that refer to external files that are portable across platforms and file systems.

In capture mode, if you select a file outside the path defined by `FORTE_AUTOTESTER_ROOT`, it is recorded with its local, literal name. The following file is an example:

```
/forte1/d/another.bmp
```

## Playing Back Captured Input

Before you can run in playback mode, you must take the following preparation steps.

### ► To prepare for playback

1. Import the input capture class (.cex) files that were created during input capture.
2. In the Project Workshop, open the PencilPlayTester project.
3. Select the component > Import Class command. The .cex files are in the directory you designated in the Capture Window.
4. Select the .cex files you wish to import; note that they will overwrite existing classes with the same name (test1 and so on).

5. Modify the LoadTestClassArray method in the MyDriver class of the PencilPlayTester project.

This method assigns the classes for tests 1 through 99 to rows in the TestClassArray array, but every line is commented out. Remove the comment characters on all the lines that correspond to class files you just imported. Note that you must always add tests to this array sequentially and starting with 1. This will enable the replay mechanism to allocate instances of each test that you want to run.

```

TestClassArray = new;

-- TestClassArray[1] = Test1;
-- TestClassArray[2] = Test2;
-- TestClassArray[3] = Test3;
-- TestClassArray[4] = Test4;
-- TestClassArray[5] = Test5;
...
-- TestClassArray[98] = Test98;
-- TestClassArray[99] = Test99;

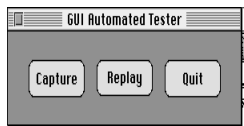
```

6. Compile and save the project.

Now you can play back one or more of the selected test runs.

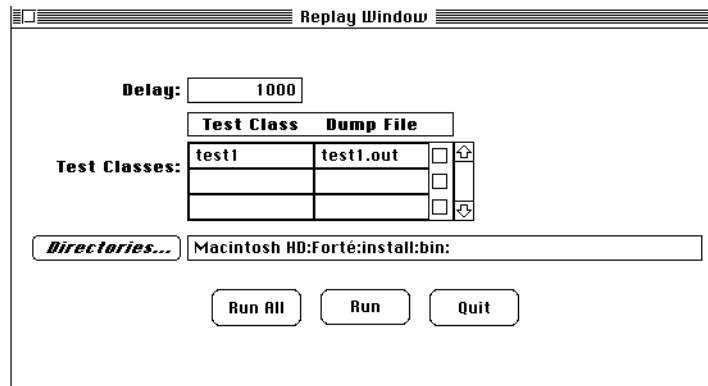
► **To play back the PencilPlay tests**

1. Run the PencilPlayTester project. The AutoTester Control Panel window appears.





- Choose the Replay button. The Replay Window, used to replay the captured input, displays as follows:



The Delay field allows you to override the setting for the environment variable `FORTE_AUTOTESTER_DELAY`. (See below for more information on this variable.)

- Click the Run or Run All buttons to play back the tests.

You can replay one test case, a group of tests (the scroll list allows multiple selections), or all the test cases in the TestDriver project.

Playback generates a dump file called `test#.out` for each test case. Set the Directories field to the directory where you want these files to go. The dump files contain descriptions of user actions and dump state information.

## Analyzing the Results

Validate the test results by comparing the dump (.out) files. You can use the dump file from any run of the PencilPlayTester project as canonical output. You can rename the canonical files (from test1.out to test1.ref, for example). You can compare the results of subsequent runs of the same test to the original output, using standard “diff” utilities for your operating system.

For each application, there may be key points in the test where you want to invoke the DumpState method to dump information about the current widget or perform other runtime validation. There may also be certain fields for which you want to avoid dumping state information, such as fields displaying current date or time data.

If you are testing on multiple platforms, use the key sequences that cause DumpState to pass the DumpWidget method its outputMask parameter as TW\_VALUE and TW\_STATE to produce the most portable dump state information.

## Automating Regression Tests

You can write scripts to run several GUI application tests with the push of a single button. The format of the .cex file makes this possible. Each .cex file contains two methods. The InputDriver method contains all the user actions on the display system. The Runit method replays those actions against the target application.

The inclusion of the Runit method in the .cex file allows you to run the .cex file using Fscript. First, you must create a tester project, following the steps described under “[Setting Up for Input Capture](#)” on page 169.

You can define the environment variable FORTE\_AUTOTESTER\_DELAY to set the PlaybackDelay attribute to the value of your choice. By default this value is 1000 milliseconds. If you set it to 0, the tests will run too quickly to be seen, but it is a good choice for batch mode runs.

You must specify the input and output files. You can use the following commands as a template for the file you input to Fscript:

```
UsePortable
CommentOn
Open
# YourTesterProject is set up like the PencilPlayTester project
FindPlan YourTesterProject
Compile %{{YOUR_VAR}}/your_path/test1.cex
SetProjStart Test1 Runit
Run
Quit
```

The results of the test are written to the dump file you specify for Fscript. Use your local operating system tools to compare the results from this run to the canonical output.

See the *Fscript Reference Manual* for information about using Fscript.

## Creating Your Own Test Utility

If you have special testing needs that are not covered by the AutoTester project, you can create your own automated testing facility using iPlanet UDS Display Library classes. In particular, see the Capture and Playback classes in the Display Library online Help. Also see the TraceWidget method in the Widget class, and the Capture, Playback, and PlaybackDelay attributes in the WindowSystem class. Review the AutoTester project for a better understanding of how you can customize and use these features.

# Using iPlanet UDS Logging Tools

This chapter describes iPlanet UDS logging tools and how you use them. Logging information from other iPlanet UDS manuals is summarized here and provided with new examples and example scenarios.

This chapter includes the following information:

- how to categorize logging functions with iPlanet UDS log filters
- how to use LogMgr methods
- how to turn on logging
- how to find the logged output

For complete information on the primary mechanism for iPlanet UDS logging—the LogMgr class—see the Framework Library online Help. System administrative tasks that use logging are more fully explained in the *iPlanet UDS System Management Guide*.

## About iPlanet UDS Logging

The most basic purpose of logging is to send output to a device. This simple function provides a window with which you can view information about an application while it is running. The information you gather can serve any number of purposes, such as debugging, gathering performance information, analyzing application flow, or changing the behavior of the program.

► **To implement logging in iPlanet UDS**

1. Decide what kind of logging information you need.

Logging is a flexible tool with a wide range of possibilities. Some brief words on how to focus your logging efforts are given in [“Logging Requirements” on page 183](#).

2. Decide how you want to filter logging information.

iPlanet UDS provides extensive and flexible filtering to allow you to gather exactly the information you need at a given run of the application. For a description of the filters (also called *log flags*), see [“iPlanet UDS Logging Filters” on page 185](#).

3. Implement your logging requirements with LogMgr methods.

You use methods on the LogMgr class to send information to log files and displays, test for or return current log flag values, and modify those flags. For a description of LogMgr methods, see [“Implementing Your Logging Scheme with LogMgr Methods” on page 190](#) or the Framework Library online Help.

4. Turn on the desired log flags and tell iPlanet UDS where to write the output.

The LogMgr methods only manipulate logging information when the corresponding flags are currently set in the environment. For information on the many approaches available for specifying what to log and where to log it, see [“Setting up Logging with iPlanet UDS” on page 196](#).

5. Modify log flag settings, if necessary, during runtime using the Environment Console or Escript **ModLogger** or **ModLoggerRemote** commands.

For more information, see [“Modifying Log Flags with Fscript and Escript” on page 199](#).

6. Test run your application.

How you test your application is influenced by the configuration of your development environment, as well as the application itself. For information on testing modes, see [“Choosing a Testing Mode” on page 209](#).

7. Examine the logging output.

Where and how you set the log flags determines where iPlanet UDS puts logging information. For information, see [“Locating Logging Output” on page 210](#).

8. Remove or disable any logging instrumentation from your application when it is of no further use.

Logging is I/O intensive. To realize the full performance potential of your application, you will probably want to remove the logging instrumentation, when you no longer need it.

## Logging Requirements

Logging provides you with tools to monitor your application. You will typically want to implement logging when you do the following:

- define internal and external interfaces of your application
- test the structural design of your application
- examine performance and flow at the “hot spots” of your application’s design

Generally, your goal is to create logging code so that the application can give reliable performance and debugging information. Since logging involves I/O, using it affects performance. Therefore you should strive to minimize the impact of your logging code on how the application runs. Understand that you will have to disable much of the code eventually to realize the full performance of the application.

Typical logging requirements are:

- tracking the flow of the application
 

You can use logging to mimic the functions of a debugger. A simple way to do this is to have each method log its name to a file whenever it is called. The resulting logging output shows the flow of the application.
- monitoring performance
 

A simple way to log performance information is to set a timer to time a database transaction, then log the result to a file. Or you can use a timer to set an acceptable time limit to a transaction duration, then log which transactions exceed the limit.
- checking state changes
 

You can use logging to monitor the changing values of a data in a variable or an array before and after significant events, much as you do with the Debugger.

- changing the behavior of a program

You can use any number of methods to alter a program based on logged information. The Test and GetLevel methods on LogMgr provide ways to change behavior based on which log filters are currently active.

## iPlanet UDS Logging Tools

The tools of the iPlanet UDS logging mechanism include:

- a hierarchy of logging filters to distinguish logging functions:
  - message types
  - service types
  - group numbers
  - level numbers

For a description of these flags, see [“iPlanet UDS Logging Filters” on page 185](#).

- a TOOL log manager class (LogMgr) with methods that do the following:
  - send logging information to a file or display (LogMgr.Put and LogMgr.PutLine)
  - modify log filters during runtime (LogMgr.Modify)
  - check current settings of log filters (LogMgr.Test)

For a description of the LogMgr class, see [“Implementing Your Logging Scheme with LogMgr Methods” on page 190](#) or the Framework Library online Help.

- options for turning on the log flags and directing the output to a file, including:
  - the FORTE\_LOGGER\_SETUP environment variable
  - the -fl iPlanet UDS command flag
  - the Log Flags page of the iPlanet UDS Control Panel
  - the Modify Log Flags command (Repository Workshop or Environment Console)
  - some system administrator Environment Console or Escript commands

For descriptions of these options, see [“Setting up Logging with iPlanet UDS” on page 196](#).



# iPlanet UDS Logging Filters

iPlanet UDS provides four levels of logging filters for organizing your logging information. Logging filters are also referred to as *log flags*. When you code logging instructions, you specify these filters as parameters of LogMgr methods. To activate logging, you specify which filters are to be active with the FORTE\_LOGGER\_SETUP environment variable, or the `-fl` command flags. For log-activating tools that have a user interface (such as the Log Flags page of the iPlanet UDS Control Panel and the Modify Log Flags command of the Repository Workshop or Environment Console), you enter these flag values in the proper fields of the window.

The iPlanet UDS logging filter types, from most general to least, are:

- message types (mandatory)
- service types
- group numbers
- level numbers

This section describes the log flag types, but not how you activate them. For setting log flags, see [“Setting up Logging with iPlanet UDS” on page 196](#).

## Message Types

The message type filter is mandatory and specifies the most general logging categories. You use message types to differentiate messages such as errors, debugging information, or performance data. The table below lists all the message types in the three different ways you specify them, namely:

- String: the string you use in command line syntax
- Label: the label you select in the Message field in the iPlanet UDS Control Panel or the Modify Log Flags window
- Constant: the runtime LogMgr constant you use with the more complex variations of the Put and PutLine methods

**Table 6-1** Message Types

String	Label	Constant	Meaning
aud	Audit	SP_MT_AUDIT	Audit messages
cfg	Configuration	SP_MT_CONFIGURATION	Configuration setting
err	Error	SP_MT_ERROR	Error messages
prf	Performance	SP_MT_PERFORMANCE	Performance information
res	Resource	SP_MT_RESOURCE	Resource information
sec	Security	SP_MT_SECURITY	Security messages
trc	Debug	SP_MT_DEBUG	Debugging information

For example, the following FORTE\_LOGGER\_SETUP specification logs trace (debugging) messages to standard output, error messages to standard output and an error log file, and performance information to a performance log file:

```
%stdout (trc:user err:user) err.log(err:user) perf.log(prf:user)
```

## Service Types

You can subdivide message types into service types to represent the large subdivisions you make within your program. Service types typically map to projects or other large portions of an application, such as inventory control, accounts receivable, or employee administration.

Service types are optional. There are 10 user-defined service types available for your use, "user1" to "user10." The table below shows the different forms of these labels, similar to the table for message types. The columns designate:

- String: the string you use in command line syntax
- Label: the label you select in the Service field in the iPlanet UDS Control Panel or the Modify Log Flags window of the Environment Console

- **Constant:** the runtime LogMgr constant you use with the more complex variations of the Put, PutLine, and Test methods

**Table 6-2** Service Types

String	Label	Constant	Meaning
user1, user2,... user10	User1, User2,... User10	SP_ST_USER1, SP_ST_USER2,... SP_ST_USER10	User-defined
user, *	User1, User2,... and User10 (use all 10)	SP_ST_USER1, SP_ST_USER2,... SP_ST_USER10 (use all 10)	All user-defined service types

## Shortcuts for Specifying all Service Types

The bottom row of table above indicates that when you specify a service type on a command line, you can use the special string “user” or an asterisk (\*) to represent all service types. However, there are no equivalents to these shortcuts when you are using the Control Panel or the Modify Log Flags window, or when you are specifying a parameter in a LogMgr method, where you must use the full enumeration.

For a service type example, you might assign different service type numbers to inventory control, accounts receivable, or personnel, or you might use different service types for different projects. The following specification sends all tracing to standard output, plus tracing from service types “user1” and “user3” to a file named trc1\_3.log:

```
%stdout(trc:user) trc1_3.log(trc:user1 trc:user3)
```

For information on the log flag syntax used in this example, see [“Setting Log Flags with Command Line Syntax” on page 201](#).

## iPlanet UDS Internal Service Types

A number of non-user service types for iPlanet UDS internal and Support use only are also visible in the drop-down list above the User1 - User10 selections of the Service field of the Log Flags page of the iPlanet UDS Control Panel or the Modify Log Flags window of the Environment Console. These types are sometimes specified as part of a release note for backward compatibility, but in general, are for iPlanet UDS internal use and should only be used with proper authorization. [“Useful Message Filters” on page 189](#) illustrates the uses for some of these service types.

## Group Numbers

You can subdivide service types into groups designated by numbers between 1 and 63, inclusive. Groups are optional, and typically map to groups of related facilities.

For example, you could subdivide service “user3” into three groups called “transactions in progress,” “queued work lists,” and “problem reports.” If “transactions in progress” is group 2, and “problem reports” is group 4, the following specification logs performance information from group 2 into one file and trace information from group 4 into another file:

```
xactprog.prf (prf:user3:2) probrep.trc (trc:user3:4)
```

The group number you specify in a Put method can be a constant that you define as equivalent to a numeric literal you specified in FORTE\_LOGGER\_SETUP. For example, assuming the literal 2 indicates the “transaction in progress” group and the value of the TOOL constant TRANSACT\_IN\_PROGRESS is 2, you could specify the following:

```
task.Part.LogMgr.PutLine(SP_MT_PERFORMANCE,
SP_ST_USER3, TRANSACT_IN_PROGRESS, 1, perfTextData);
```

For a description of the PutLine method, see the iPlanet UDS online Help.

You can also specify a range of group numbers using the syntax *group#-group#*. In the previous example, if you want trace information from groups 2 through 4 to go to a specific file, you would use the following statement:

```
some_trc.log(trc:user3:2-4)
```

## Level Numbers

You can subdivide groups into levels designated by numbers from 1 to 255, inclusive. Levels are optional, typically used to specify particularly detailed levels of information. The greater the level number value, the more detailed the information.

As with group numbers, the level number is determined by the application. Typically, developers use level numbers to filter out trace messages. The following example indicates that all level 1 trace data from group 2 (transactions in progress) of the “user3” service should be printed to standard output.

```
%stdout (trc:user3:2:1)
```

Given the log setting above, the following fragment prints only one line:

```
log: LogMgr = task.Part.LogMgr;
-- Printed (level >= 1)
  log.Put(SP_MT_DEBUG, SP_MT_USER3, TRANSACT_IN_PROGRESS, 1,
    'Browsing account # ');
  log.PutLine(SP_MT_DEBUG, SP_MT_USER3, TRANSACT_IN_PROGRESS,
    1, acc.Number);
-- Not printed (level >= 2)
  log.Put(SP_MT_DEBUG, SP_MT_USER3, TRANSACT_IN_PROGRESS,
    2, acc.Owner);
  log.Put(SP_MT_DEBUG, SP_MT_USER3, TRANSACT_IN_PROGRESS,
    2, acc.LastChangeDate);
```

## Useful Message Filters

The following table describes a number of iPlanet UDS runtime system message filters you might find useful in diagnosing system management problems:

**Table 6-3** Useful Message Filters

Filter	Function
trc:cm:*:4	This filter is most useful if there are communication problems when first setting up your environment.
trc:db:1-8	Used to diagnose problems in accessing a database.
trc:lo:25	Always set this filter when tracking down problems, otherwise key exceptions may not be displayed. However, most users should not have this filter set because they may be alarmed at the number of harmless exceptions logged.
trc:os:1:1	Mostly used by developers to track object memory requirements. Logs automatic memory management activities.
trc:os:4:5	Mostly used by developers to track object memory requirements. Used with trc:os:1:1 to collect information on objects.
trc:os:5:5	Mostly used by developers to track object memory requirements. Used with trc:os:1:1 to collect information on pages and objects.
trc:os:10	When this filter is set, the value of environment variables will be logged when any iPlanet UDS process is started.

**Table 6-3** Useful Message Filters (*Continued*)

Filter	Function
trc:os:14	Used to show dynamically loaded libraries.
trc:rp:2:50	Used to show user operations on the local client that can affect the repository. trc:rp:2:75 provides more detailed information.

## Implementing Your Logging Scheme with LogMgr Methods

At the heart of iPlanet UDS logging is the LogMgr class. iPlanet UDS creates a LogMgr object each time a TOOL program is invoked. You use methods on the LogMgr object to control the logging information that results from the execution of a TOOL program.

This section presents a summary of LogMgr class methods. For a complete description of the LogMgr class, see the Framework Library online Help.

This section discusses how to:

- reference the LogMgr object
- send information to log files or displays using the Put, PutLine, PutHex, and PutHexLine methods
- alter the flow of control of an application using the Test and GetLevel methods
- change log filters using the ModifyFlags method
- flush current log files to disk using the Flush method

The term “current logging flag settings” means the values that are current in the running environment, namely, the values established for the iPlanet UDS process at the time the process was started. Environment variables changed after the process is started are not picked up by the process. For information on the mechanisms for setting environment variable values and their order of precedence, see [“Setting up Logging with iPlanet UDS” on page 196](#).

## Referencing the LogMgr Object

Never create an object of type `LogMgr`; instead, reference `task.Part.LogMgr`. (The `task` key word references the `TaskHandle` object; the `Part` attribute is a read-only `TaskHandle` attribute that contains the current `Partition` object for the task. For information on the `TaskHandle` class, see [Framework Library online Help](#).) For example, to print messages to the trace text display, you can use statements such as:

```
task.Part.LogMgr.PutLine('Initializing analysis screen...');
...
task.Part.LogMgr.PutLine('Closing analysis screen...');
```

You may find it more convenient to declare a variable of type `LogMgr` and assign to it the value of the global `LogMgr`, as follows:

```
log: LogMgr = task.Part.LogMgr;
log.PutLine('Initializing analysis screen...');
...
log.PutLine('Closing analysis screen...');
```

## Logging Application Information with PutLine

When logging, you can set messages to be displayed that reflect the state of your application or to add information about an exception that occurred while your application was running. This is especially useful when a user receives a critical error message displayed in a user window and you want to write more details about the error in a log file.

You use the `PutLine` method, or one of its related methods, to send information to log files or displays. The `PutLine`-related methods are:

- `Put`
- `PutLine`
- `PutHex`
- `PutHexLine`

Both Put and PutLine methods are overloaded. One variation is used to log message text; the other is used to log state information. PutHex and PutHexLine have only one variation, and are used to log message text that may be very large integer values or machine addresses of object references in hexadecimal format.

## Logging Message Text

To write textual values to all logging windows and log files, call the simple Put or PutLine method. This variation converts its source input parameter to a text value and prints it to all log files (if the FORTE\_LOGGER\_SETUP or **-fl** flag is set).

## Logging State Information

To write log messages that reflect the state of your application, you can add your own trace flags to your application with the complex variation of the Put or PutLine methods. These methods can flag an area in your application to generate logging information when the custom trace flags are turned on. To log messages for an application area, you set the message filters the same as the parameters. The log message is written only to the log files satisfying the filter setting.

## Put and PutLine Examples

The Put and PutLine methods allow you to filter out certain messages according to their level of detail or type. For example, the following fragment is printed only if you specified its level in the current logging flag setting.

### Code Example 6-1 Filtering Message

```
event loop
  ...
  when <SaveButton>.click do
    task.Part.LogMgr.PutLine(SP_MT_DEBUG,
      SP_ST_USER1,
      EMP_DEBUG_GROUP,
      5, 'Saving employee data');
    self.Save();
  ...
end event;
```



This example specifies that the string “Saving employee data” is logged if the following conditions are true:

- debugging is on for User1 Service
- debugging is on for the employee module debugging group (defined through a user-defined integer constant EMP\_DEBUG\_GROUP)
- the level of detail is at least 5

If, at runtime, the settings specified in the above PutLine method invocation are not included in the settings specified in the current logger flag settings, then the message is not printed.

The following example assumes that current logger flag settings have been set to log debugging messages (for example, “%stdout(trc:user)”). The Put and PutLine methods use the simpler variation and do not filter any message, service or group type, or detail level. When a user clicks the TraceEmpInfo button, the program displays a detailed description of the current employee object to all trace log output files (including the trace window).

#### Code Example 6-2 Logging Debugging Message

```
log: LogMgr = task.Part.LogMgr;
event loop
...
when <TraceEmpInfo>.Click do
  log.Put('Employee # '); log.PutLine(emp.Num);
  log.Put(' Name  : '); log.PutLine(emp.Name);
  log.Put(' Age   : '); log.PutLine(emp.Age);
...
  log.PutLine(' Review History: ');
for review in emp.ReviewHistory do
  log.Put('     Review by: '); log.PutLine(review.MgrName);
  log.Put('     On    : '); log.PutLine(review.Date);
  log.PutLine(review.Text);
end;
...
end event;
```

## Altering the Flow of Control of an Application

You can use the `GetLevel` and `Test` methods to alter the flow of control of an application or to alter debugging calls in your TOOL code. `Test` allows you to check whether specific logger flags are currently set in the environment; `GetLevel` lets you check what level is set, given a specific message, service, and group setting. See the iPlanet UDS online Help for detailed information about the `GetLevel` and `Test` methods.

### Test Method Example

The following code tests the `LogMgr` object for a particular performance setting that could have been set using either `FORTE_LOGGER_SETUP` or the `ModifyFlags` method.

#### Code Example 6-3 Testing LogMgr Settings

```

if task.Part.LogMgr.Test(SP_MT_PERFORMANCE,
    SP_ST_USER1, ACCOUNT_SCAN_RATE, 1) then
    self.CollectStatistics = true;
end;
...
event loop
    when <RetrieveAccount>.Click
        account = self.RetrieveAccount(accountNum);
        if account <> nil and self.CollectStatistics then
            -- Track the number & name of all accounts retrieved.
            self.Retrieved = self.Retrieved+1;
            self.AccountsRetrieved[self.Retrieved] = account;
        end if;
        ... -- Do the same for accounts scanned, corrected, etc.
    end event;
if self.CollectStatistics then
    task.Part.LogMgr.Put(SP_MT_PERFORMANCE,SP_ST_USER1,
        ACCOUNT_SCAN_RATE, 1,'List of accounts scanned by ');

    task.Part.LogMgr.PutLine(SP_MT_PERFORMANCE,SP_ST_USER1,
        ACCOUNT_SCAN_RATE, 1,userName);
    ...
end if;

```

## Changing Logging Filters

The `ModifyFlags` method allows you to specify which messages are to be logged while the application is running. The `ModifyFlags` method is overloaded. One variation is based on modifying a source parameter; the other variation is based on log flag specifications. See the iPlanet UDS online Help for detailed information about the `ModifyFlags` method.

## Flushing Current Log Files

By default, iPlanet UDS flushes output to `stdout` immediately. However, if you redirect logging output to additional files, iPlanet UDS uses the native operating system's usual file buffering, which batches the output in approximately 1K blocks, depending on the particular operating system. This is of particular importance to client partitions, because their `stdout` is sent to the Launch Server window (except on UNIX, where the `stdout` for the Launch Server is sent to the `ftlaunch` log). Therefore, if you wish to direct any of the output into a file, you need to flush all the output to ensure the information is logged immediately.

There are three ways to guarantee that all logging output is sent out to the log files:

- Explicitly flush output using the `LogMgr.Flush` method (described in the iPlanet UDS online Help).
- Issue the **FlushLogFiles** command on the active partition agent of interest.  
See the section on logging and log files in the *iPlanet UDS System Management Guide* for specific information on log file buffering with the **FlushLogFiles** command.
- Gracefully exit the Launch Server by using the **ftcmd shutdown server** command or shut down the Launch Server using `Escript`.

If you quit the Launch Server (rather than exiting it as described above), it will not flush to the log files.

To navigate to the Launch Server's agent in `Escript`, use the following commands:

```
findactenv
findsubagent nodename
findsub ftlaunch_c10_client
showag
```

At this point, you are at the Launch Server and the active instances of it on the given node are listed, such as:

```
Current Agent :
  Type:         Installed Partition
  Name:         FTLaunch_cl0_Client_HENNA
  Status:       ONLINE
  Parent Agent : Node Agent - HENNA (ONLINE)
  Instruments:  None
  Sub Agents:
    Active Partition Agent - FTLaunch_cl0_Client_0x113:0x1 (ONLINE)
escript >
```

Type exit here at the Escript prompt to exit the Launch Server.

See the *Escript and System Agent Reference Manual* for information on Escript commands.

## Flush

See the Auction example in “[Logging Examples](#)” on page 214 for an illustration of using the Flush method.

# Setting up Logging with iPlanet UDS

There are several ways to set up message logging for your application. Some are designed for use by developers during development sessions, and some are for system managers managing development or deployment environments. Though this chapter is directed at the developers of applications, system management tools are included, because testing an application takes place in a distributed environment.

This section describes how to set logging parameters on the command line and with the graphical interface tools.

## Tools for Setting Log Flags

This section describes the many ways to set log flags.

### The FORTE\_LOGGER\_SETUP Environment Variable

The FORTE\_LOGGER\_SETUP environment variable sets the default log flags and output file for a given node.

iPlanet UDS sets the default settings during installation. To modify the default settings, use your platform's procedure and syntax for setting environment variables.

---

**NOTE** We strongly urge you to consult the section on how to set environment variables without the iPlanet UDS Control Panel in either *A Guide to the iPlanet UDS Workshops* or the *iPlanet UDS System Management Guide*. Those sections provide detailed platform-specific procedures for setting environment variables. In cases where multiple procedures are available, the order of precedence is given. The order of precedence is crucial for determining what settings are current at the time of your test.

---

See also [“Precedence Details on NT” on page 200](#), for an important point to understand about precedence on NT.

### The Log Flags Page of the iPlanet UDS Control Panel

The iPlanet UDS Control Panel allows you to view the iPlanet UDS environment variable settings from all platforms. For Windows, the Control Panel allows you edit the settings, as well. Log flag settings that you edit with the Control Panel override the log flag settings you set with the FORTE\_LOGGER\_SETUP environment variable.

On Windows, when you change the settings in the Control Panel, iPlanet UDS edits the setting in the iPlanet UDS environment variable file. There may be several files from which you can set environment variables, but the file that specifically relates to the Control Panel is:

Platform	Environment Variable File
NT	Registry Editor's CurrentUser Tree file HKEY_CURRENT_USER/Software/ForteSoftwareInc/Forte

The Control Panel does not allow you to specify log file names. Therefore, the flags you set with the Control Panel are logged in the default log file, “stdout.” For clarification on the location of stdout, see [“Locating Logging Output” on page 210](#).

## The -fl Flag of iPlanet UDS Commands

The **-fl** command-line flag overrides the FORTE\_LOGGER\_SETUP and Control Panel settings for the partition in which you issue the command using the flag.

You can run any iPlanet UDS system management service or iPlanet UDS system application, such as the Environment Console, Escript, and **ftexec**) with the logger flag (**-fl**) option.

For a complete list of iPlanet UDS commands and a summary of command syntax, see the *iPlanet UDS System Management Guide*.

## The Utility > Modify Log Flags Command

If you wish to change the default filter settings at any point during your development session, you can use the Utility > Modify Log Flags command in the Repository Workshop. The command opens a window, where you view and change the filter settings in an array field.

Like the iPlanet UDS Control Panel, the Modify Log Flags dialog does not allow you to specify a target log file.

For details on the Modify Log Flags command of the Repository Workshop, see *A Guide to the iPlanet UDS Workshops*.

## The Component > Properties Command

System managers use the Component > Properties command in the Environment Console to set log flags for a partition in an application.

When started, the partition will overwrite its existing log files. To preserve the old log file, either back it up or change its name before you start the partition.

For information on specifying logger flags for a partition, see the *iPlanet UDS System Management Guide*.

## The Component > Modify Log Flags Command

System managers use the Component > Modify Log Flags command to dynamically change the current message filters for an active application or Node Manager partition.

The modified filters apply to the first log file specified in either the `FORTE_LOGGER_SETUP` environment variable or the `-fl` logger flag in the startup command for the given partition.

For details on modifying logger flags, see the *iPlanet UDS System Management Guide*.

## Modifying Log Flags with Fscript and Escript

As you can with other iPlanet UDS commands, you can specify logging parameters with the **fscript** and **escript** commands. When you specify logging parameters when starting **fscript** and **escript**, you are affecting logging for the Fscript or Escript *session*. Within the session, you can modify logging filters dynamically, using the **ModLogger** and (with Escript) **ModLoggerRemote** commands. The difference between these commands is:

- **ModLogger** modifies log flags for the current Fscript or Escript session
- **ModLoggerRemote** modifies log flags for agents you have started with the Escript tool

### *Note on the “remote” in “ModLoggerRemote”*

When you are in an Escript session, navigating the application agents of your running application, your Escript session is the agent managing that partition, but is not really on that partition itself. Therefore, the “remote” in **ModLoggerRemote** refers to the fact that you are modifying logger flags elsewhere (on the partition), not in the current process (Escript).

When you use the **ModLogger** and **ModLoggerRemote** commands to modify logger flags, you can change only the log output to the `stdout` specification. You cannot change the log flags that write to any log file that you might have specified with the `-fl` flag or the `FORTE_LOGGER_SETUP` variable. (Note that “`stdout`” for a server partition is its log file, because it has no terminal window associated with it. See [“Setting up Logging with iPlanet UDS” on page 196](#) for clarification about output log specifications.)

## The LogMgr.ModifyFlags Method

Flags set programmatically with the `LogMgr.ModifyFlags` method change current settings as specified within the application code.

For further information, see the iPlanet UDS online Help.

## Order of Precedence for Log-Setting Procedures

Knowing which log flag setting is in effect can be confusing. The following list gives the basic order of precedence for the various ways you can set log flags:

### 1. Flags set during runtime

These include flags set with the **Modify Log Flags** command of Environment Console, the **ModLogger** or **ModLoggerRemote** commands, or flags modified programmatically with `LogMgr.ModifyFlags` method.

Flags set during runtime override all other settings until you exit iPlanet UDS (including the Launch Server and the Environment Manager) or reset them using one of these commands or procedures.

### 2. Flags set for a specific process

These include flags set with the logger flag (`-fl`) command option or the Environment Console **Component > Properties** command.

### 3. Flags set with the iPlanet UDS Control Panel

This applies to Windows platforms only, because the Control Panel is read-only for all other platforms.

### 4. Flags set with the `FORTE_LOGGER_SETUP` environment variable

See the section on setting environment variables without the iPlanet UDS Control Panel in *A Guide to the iPlanet UDS Workshops*.

## Precedence Details on NT

The order of precedence given above (specifically of items 3 and 4) assumes you have set `FORTE_LOGGER_SETUP` in the NT Registry. Since the iPlanet UDS Control Panel also sets values in the NT Registry, the new values of log flags set with the iPlanet UDS Control Panel changes the previous value set for `FORTE_LOGGER_SETUP` in the Registry.

However, it is also possible to set environment variables in the NT Control Panel. Since settings in the NT Control Panel override the settings in the Registry, they override settings made with the iPlanet UDS Control Panel, giving item 4 higher precedence than item 3.

Therefore, we recommend that you do *not* use the NT Control Panel to set your environment variables. Setting environment variables in too many places can make your setup confusing and inconsistent.



## Setting Log Flags with Command Line Syntax

You use command line syntax for setting logger flag parameters using the following:

- the FORTE\_LOGGER\_SETUP environment variable
- iPlanet UDS commands
- the Component > Properties command in the Environment Console

The Fscript and Escript **ModLogger** and (Escript only) **ModLoggerRemote** commands also use *log\_flags* syntax, but cannot specify output files.

### Log Flag Syntax

The command line syntax for setting logging parameters is:

#### *Portable syntax*

*iPlanet UDS\_command* [-fl *file\_name(log\_filter)*][*file\_name(log\_filter)...*]

#### *OpenVMS syntax*

**VFORTE** *iPlanet UDS\_command*

[/LOGGER=*file\_name(log\_filter)*][*file\_name(log\_filter)...*]

#### *File Name Parameter*

The *file\_name* parameter is the valid file name of the file in which you want to log messages. The special file names “%stdout” and “%stderr” log the messages to standard output or standard error, respectively.

You can specify several files for logging messages. Multiple logging files are useful, for example, in an application where you want to display general tracing on standard output (%stdout), but want detailed tracing logged to a file for later review.

On Windows only, you can use the name “%stdwin” to create a simple, scrollable output window for textual output. “%stdwin” is particularly useful to specify an alternative file for the output from Fscript or the iPlanet UDS Workshops.

**File Naming Conventions** You must specify file names using the format native to the system on which you invoke an iPlanet UDS command. The following example shows how you could specify a file named “testlog.log” in the My\_Project/log directory on different platforms:

**Table 6-4** Syntax for Specifying Log Files

Operating System	Specifying an example log file
Windows	c:\My_Project\log\testlog.log
UNIX	/My_Project/log/testlog.log
VMS	[My_Project.log]testlog.log

### *Log Filter Parameter*

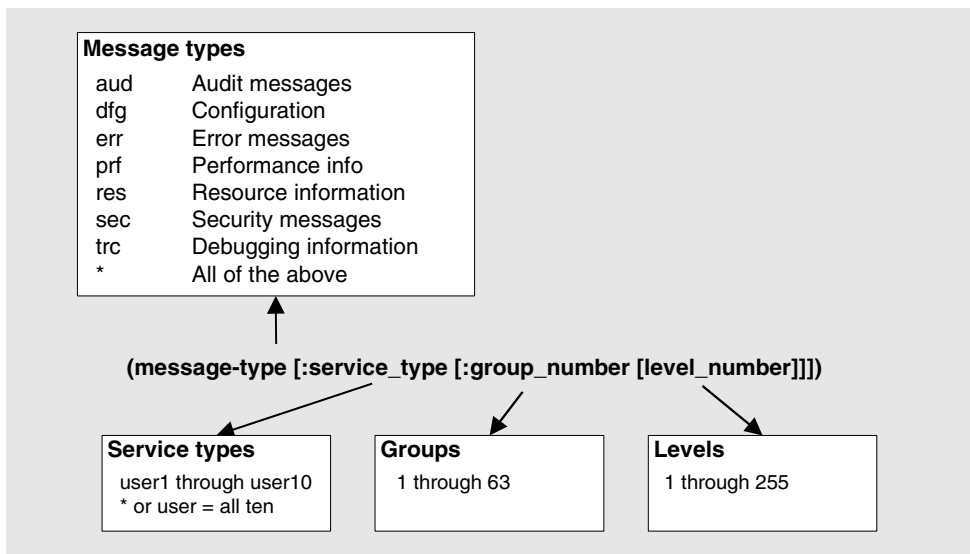
Each file name is associated with a *log\_filter* whose syntax is as follows:

#### *Syntax*

*message\_type[:service\_type[:group\_number[:level\_number]]]*

Arguments that contain parentheses must be enclosed by double quotes.

**Figure 6-1** offers a visual clarification of log filter syntax. For a description of each log filter option, see the previous section, “iPlanet UDS Logging Filters” on page 185.

**Figure 6-1** Log Filter Syntax

## FORTE\_LOGGER\_SETUP Example

This section shows how to set the FORTE\_LOGGER\_SETUP environment variable, given the following specification:

- file name: FORTE\_ROOT/log/testlog
- message type: debugging (trc)
- service type: user1
- group number: 1
- level number: 1

Use the following syntax:

**Table 6-5** Platform-specific Syntax for FORTE\_LOGGER\_SETUP

Platform	Syntax
Windows	FORTE_LOGGER_SETUP "FORTE_ROOT\log\testlog(trc:user1:1:1)"
UNIX	setenv FORTE_LOGGER_SETUP "\$FORTE_ROOT/log/testlog(trc:user1:1:1)"
OpenVMS	DEFINE FORTE_LOGGER_SETUP "FORTE_ROOT:[LOG]TESTLOG(TRC:USER1:1:1)"

The table below shows the file where you set the FORTE\_LOGGER\_SETUP variable (or logical name on OpenVMS).

**Table 6-6** Platform-specific Location for Setting FORTE\_LOGGER\_SETUP

Platform	Location of FORTE_LOGGER_SETUP
Windows NT	Machine-wide: Registry Editor's HKEY_CURRENT_USER/Software/ForteSoftwareInc/Forte file  User-specific: Registry Editor's HKEY_LOCAL_MACHINE\SOFTWARE\ForteSoftwareInc\Forte <i>version_number</i> file
UNIX	C-Shell: FORTE_ROOT/fortedef.csh file  Bourne-Shell: FORTE_ROOT/fortedef.sh file
OpenVMS	Machine-wide: FORTE_ROOT:[INSTALL.SCRIPTS]FORTE_LOGIN.COM file  User-specific: FORTE_ROOT:[INSTALL.SCRIPTS]SITE_LOGIN.COM file

## Command-Line Log Flags Example

This example shows how to start a standard client partition of iPlanet UDS on the command line, given the following specification:

- file name: FORTE\_ROOT/log/my\_log.log
- message type: debugging (trc)
- service type: user1
- group number: 3
- level number: 1-5

### *Windows Platforms*

To use flags with an iPlanet UDS command on a Windows platform, you must edit the properties of the appropriate icon.

**Windows NT** You edit the shortcut of either the iPlanet UDS Standalone (ftexec.exe) command or of the iPlanet UDS Distributed (ftcmd.exe) command.

#### ➤ **To edit a shortcut icon and start iPlanet UDS**

1. Select the appropriate shortcut in C:\Winnt\Profiles\user\Start Menu\Programs\Forte.
2. Choose the File > Properties command.

The Properties command opens the properties dialog for the shortcut. The Target field contains the fcmd (or ftexec) command line.

3. Add the following syntax to the Target command line:

```
-fl "FORTE_ROOT\log\my_log.log(trc:user1:3:1-5) "
```

4. Click OK to close the Properties dialog.
5. Start iPlanet UDS by selecting the shortcut icon.

### UNIX Platform

► **To run the iPlanet UDS command with log flags**

1. Enter the following string in a shell window.

```
ftexec -fl "FORTE_ROOT/log/my_log.log(trc:user1:3:1-5)"
```

### OpenVMS Platform

► **To run the iPlanet UDS command with log flags**

1. Enter the following string on the command line.

```
VFORTE FTEXEC
/LOGGER="FORTE_ROOT:[LOG]MY_LOG.LOG(TRC:USER1:3:1-5)"
```

## Setting Log Flags with a Window Interface

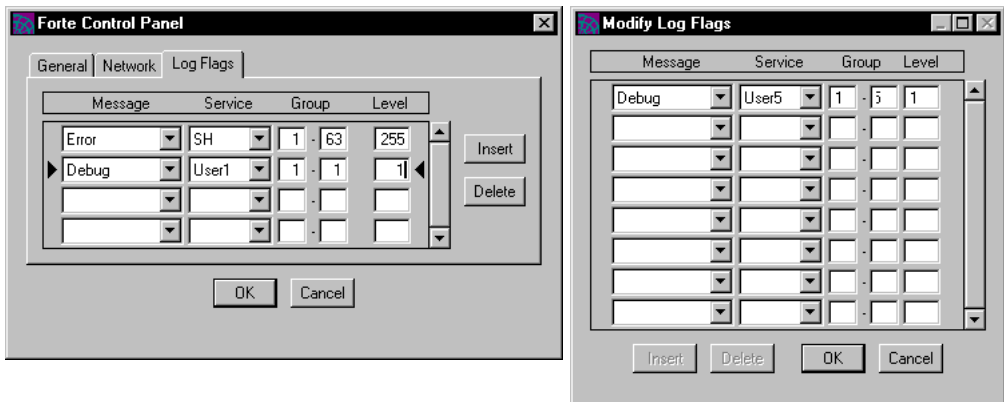
Two tools provide a user interface (a window) for specifying iPlanet UDS log flags:

- the Log Flags page of the iPlanet UDS Control Panel (Windows only)
- the Modify Log Flags dialog (raised by the Utility > Modify Log Flags command of the Repository Workshop and the Component > Modify Log Flags command of the Environment Console)

The Control Panel and Modify Log Flags dialog allow you to specify log flags, but not output files.

Figure 6-2 shows the Control Panel and Modify Log Flags windows.

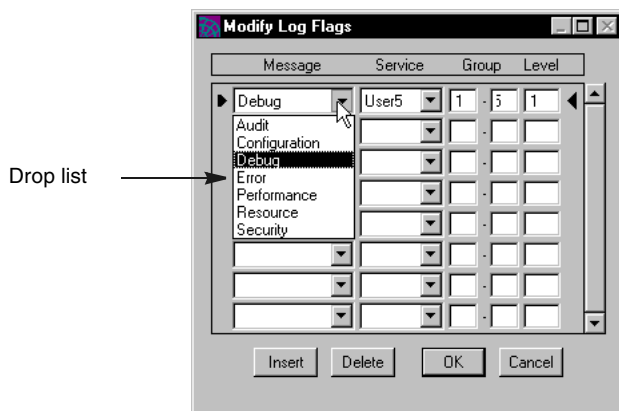
**Figure 6-2** The Log Flags Page of the Control Panel and the Modify Log Flags Dialog



## Specifying Message and Service Types

For the Message and Service types, you can either type the value directly into the field, or use the drop list as shown in [Figure 6-3](#).

**Figure 6-3** Using a Drop List for the Message Type



In command-line syntax, the following two specifications are equivalent:

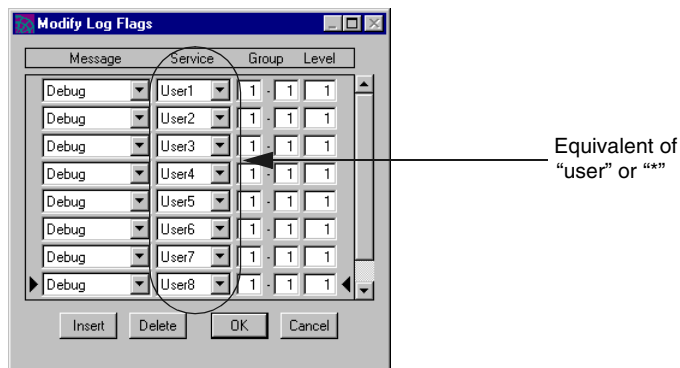
```
trc:user:1:1
```

```
trc:*:1:1
```

Both use the shortcuts “user” and the asterisk wildcard (\*) to represent all Service types (User1 to User10). There is no equivalent to these shortcuts in the Control Panel or Modify Log Flags windows. To specify all 10 service types, you must create 10 individual rows with the same Message, Group, and Level settings for Service types User1 to User10.

[Figure 6-4](#) illustrates how to specify the equivalent of the command-line shortcuts.

**Figure 6-4** Specifying All Service Types



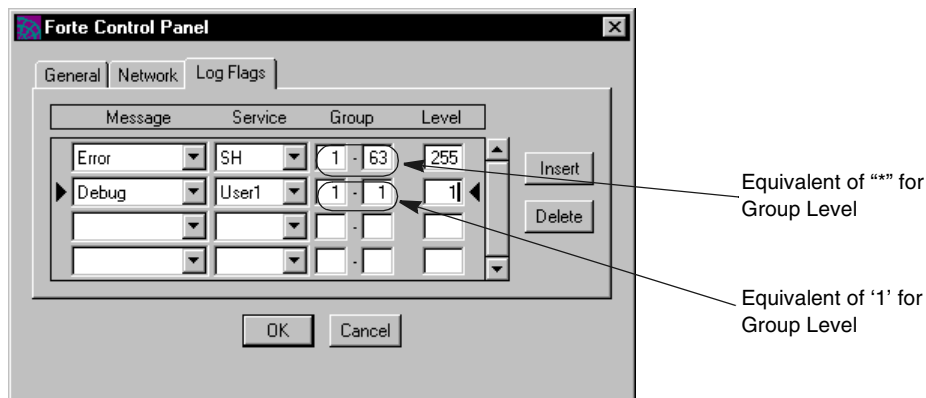
### Specifying Group Levels

In command-line syntax, you can also use an asterisk wildcard (\*) to represent all group levels. On the window, group levels have two boxes with a dash between them, so that you can easily specify a range. To specify “all groups,” specify “1-63,” as shown in [Figure 6-5](#).

To specify a single group level, specify a “range” of 1-1.

[Figure 6-5](#) illustrates how to specify all or only one group.

**Figure 6-5** Specifying All or a Single Group Level





# Choosing a Testing Mode

You can test your logging code, or use it to test parts of your application, while you are developing your application. You can test in the following modes:

- local testing
- distributed testing

The configuration of your development environment influences the testing mode you can use. For example, if you are running iPlanet UDS Standalone, local testing is the only option, and all log files will be on your local machine. If you are in a distributed environment that includes developer clients and test server nodes, both testing options are available. If you have access to the Environment Console, you have more options for modifying and reading server log files than otherwise.

## Local Testing

You test your application locally by using the Run command either from the Repository or Project Workshops. This is useful for testing your GUI windows before you create service objects. It is also advantageous when your machine does not have access to a network. You can test locally from either iPlanet UDS Standalone or iPlanet UDS Distributed.

If you are running standalone, the application is not partitioned; it runs entirely on the local machine.

If you are running distributed, iPlanet UDS places only service objects that require a server-resident resource on the server. For example, a service object that contains a DBResourceMgr or DBSession service object may run on the server if the client machine does not contain the database resource manager. All other objects are placed on the client.

## Distributed Testing

You test your application in distributed mode by using the Run command from the Partition Workshop. If you have already partitioned your application at least once, you can also test distributed by using the Run Distributed command from the Project Workshop.

Distributed testing is used to test the operation and functionality of an application distributed across multiple nodes, though it is possible to have all “distributed” partitions all on the same iPlanet UDS node or on multiple nodes on the same physical machine. You must use iPlanet UDS distributed to test in this mode.

### Distributed Testing and Test Service Object Names

Multiple testers can test the same application simultaneously, because iPlanet UDS creates unique instances of all the application’s services for each instance of the application.

## Locating Logging Output

When you set log flags where you can also specify an output file, knowing where the logging output will be written is simple: it is written to the specified log file. If you specify log flags in the iPlanet UDS Control Panel or the Modify Log Flags window, you know only that the generated output is written to “standard output” (or “stdout”) on the node of the partition where you set the flags. What is “standard output” exactly?

Each iPlanet UDS partition has a default log file, called standard output, whose name depends on the iPlanet UDS partition for which logging is being performed. For example:

- on a client partition, standard output is usually a window
- on other partitions, standard output is a file, whose name depends on the type of partition

Standard Output names are listed in the table below:

**Table 6-7** Standard Output

iPlanet UDS Partition	Standard Output Log File in FORTE_ROOT/log/
Client partition	Windows: Launch Server window UNIX: <i>ftlaunch_port.log</i> , where port is the socket number for the Launch Server. OpenVMS: The DECterm window from which you started iPlanet UDS.
Active standard partition	<i>forte_ex_process_ID.log</i> (for example, <i>forte_ex_13456.log</i> )

**Table 6-7** Standard Output (*Continued*)

iPlanet UDS Partition	Standard Output Log File in FORTE_ROOT/log/
Active compiled partition	<i>filename_process_ID.log</i>
Launch Server	Windows: Launch Server window UNIX: <i>ftlaunch_port.log</i> , where <i>port</i> is the socket number for the Launch Server. OpenVMS: there is no Launch Server.
Node Manager	<i>node_name.log</i> , where <i>node_name</i> is the first 8 characters of the node name.

## Changing Log File Names for Active Partitions

You can change the default log file name for an Active Partition agent. Changing the default log file name makes iPlanet UDS close the current log file and open a new file with the name you specify.

You change the default log file name of an active partition using the Environment Manager. Please consult the *iPlanet UDS System Management Guide* for a complete description of the Environment Manager and background information for the steps for changing log file names given here.

**NOTE** You can only specify log file names for Active Partitions agents for compiled server partitions and iPlanet UDS executor partitions. Client partitions can only log information to trace windows.

You can change the log file name for an Active Partition agent by navigating to its agent and opening the Instruments window. You can then either change the instrument logging options by using the Instrument Logging Properties dialog or by directly changing the values of the instruments that control these options.

## Rules for Log File Names

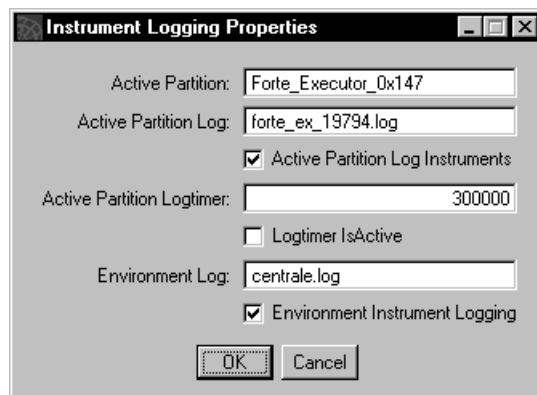
When you specify the new name of the log file, you need to use a portable file name syntax (UNIX style). If the Log Filename does not start with a */*, then the file is given relative to the FORTE\_ROOT/log directory on the node on which the service is executing. If the Log Filename does start with a */*, then it specifies an absolute path on the node on which the service is running.

The steps described in this section for changing the log file names using the Environment Console correspond to changing the values of the LogFile instrument for the Active Partition agent. For information about changing instrument values, see the *Esript and System Agent Reference Manual*.

The steps for changing the log file name are different depending on whether you are changing the log file name for a compiled active partition or an interpreted active partition. Although all active server partitions can log data to log files, compiled partitions each have their own log files, while interpreted partitions log data to the log files of their instances of the iPlanet UDS interpreter (iPlanet UDS Executor). Therefore, to change the log file name for an interpreted partition, you need to change the log name for the active partition of its iPlanet UDS Executor instance, as described below.

► **To change a log file name for a compiled active partition or iPlanet UDS executor partition**

1. In the Environment Console's Active Environment window, select the View > Node Outline command.
2. Find the compiled Active Partition agent whose log file name you want to change by expanding the browser outline view.
3. Open the Agent window for the active partition by selecting the agent, then choosing the Component > Open command.
4. Open the Instruments window for the current agent by choosing the File > Instruments command in the current Agent window.
5. Open the Instrument Logging Properties dialog by choosing the File > Instrument Logging command.



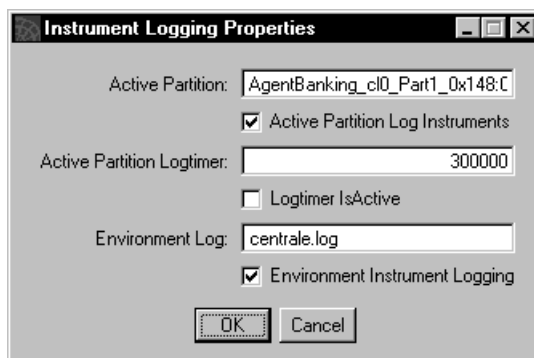
6. In the Instrument Logging Properties dialog, enter the new log file name for the active partition's log file in the Active Partition Log field. You can also change the name of the Environment Manager's log file by entering the new name of the log file in the Environment Log field.

► **To change the log file name for an interpreted active server partition**

1. In the Environment Console's Active Environment window, choose the View > Node Outline command.
2. Locate the Forte\_executor\_nodename agent, which is a subagent of the Node agent for the node where the standard partition is running.
3. Click the expansion arrow next to the Forte\_executor\_nodename agent, then double-click the Active Partition agent whose name matches the name of the standard partition.

The Agent window for the Active Partition agent opens.

4. Open the Instruments window for the Active Partition agent by choosing the File > Instruments command of the current Agent window.
5. Open the Instrument Logging Properties dialog by choosing the File > Instrument Logging... command the Instruments window.



6. In the Instrument Logging Properties dialog, enter the new log file name for the active partition and the instance of the iPlanet UDS executor in the Active Partition Log field. You can also change the name of the Environment Manager's log file by entering the new name of the log file in the Environment Log field.

# Logging Examples

This section provides two logging examples. The first, the Auction example distributed with your iPlanet UDS software, shows a simple way to log application flow. The second shows how to set up methods to time themselves and send the results to another method that logs the data in three different formats, depending on what log flags are set.

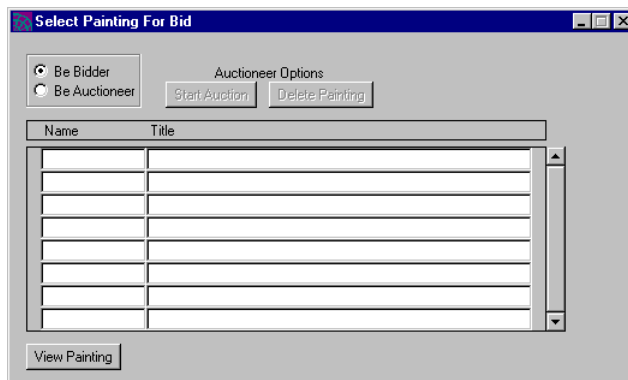
## Auction Example

The Auction example included in your iPlanet UDS distribution offers a simple example for logging that follows application flow. See [Appendix A, “iPlanet UDS Example Applications”](#) for information on how to set up and run the example.

The application provides a list of paintings available for bidding. Bidders located at their respective computers can bid on paintings being offered by an auctioneer located at some other computer. Bidders bidding on the same painting are notified when the someone has changed the price by placing a higher bid.

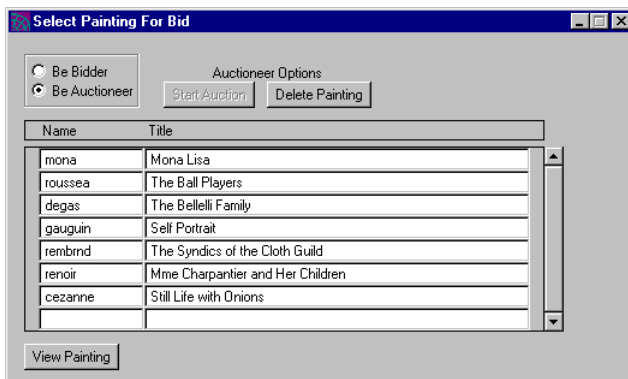
When you first start the Auction application, a window opens displaying an empty array field for listing artists' names and painting titles, radio buttons to select Bidder or Auctioneer role, Auctioneer options to start the auction or delete a painting, and a button to view the painting, as shown below.

**Figure 6-6** Auction Example Starting Window



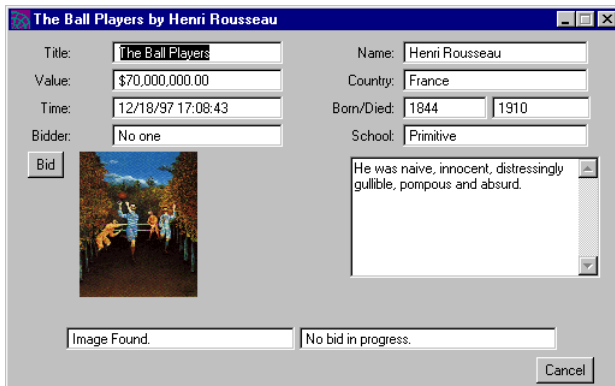
The Auctioneer starts by selecting the Be Auctioneer toggle, then clicking the Start Auction button, which fetches the list of artists and their paintings and displays them in the array field, as shown in [Figure 6-7](#). At any time, the Auctioneer can delete paintings from the array.

**Figure 6-7** Paintings Listed in the Starting Window

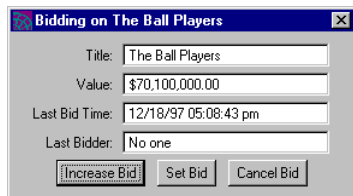


The Bidder action starts by selecting the Be Bidder toggle, then selecting a painting, and clicking the View Painting button. This displays a view window with a description of the painting, an image of the painting, a Bid and a Cancel button, shown in [Figure 6-8](#).

**Figure 6-8** Auction Example View Window



Clicking the Bid button brings up the Bid window, showing title of the painting, the current value, and the time of the last bid and the name of the bidder issuing it. The Bid window is shown in [Figure 6-9](#).

**Figure 6-9** Auction Example Bid Window

The bidder can then increase the bid, set the bid, or cancel the bid. While the bid window is up for one bidder, other bidders bidding on the same painting are informed on their View Painting window that a bid is in progress.

## Setting up Logging

To follow application flow, the logging strategy in the Auction application uses a simple scheme, consisting of:

- a single log flag setting (trc:user1:1:1) to turn logging on
- a test if the log flag is set, then a PutLine call to log the data pertinent to the method
- flushing all logging after the PutLine method is called

The strategic actions that are logged are:

- when the Auctioneer/Bidder roles are selected
- when the artist/painting array field is filled
- when a painting is deleted from that array field
- when a bid starts and ends



The application is comprised of four projects: Auction, AuctionServerProject, ImageProject, and Utility. Methods that perform the strategic actions exist in two of these projects. The table below lists these projects, the classes and methods that include the logging, and the corresponding .pex files, where you can find the complete code. (Example .pex files are found in subdirectories of the FORTE\_ROOT/install/example directory.)

Project	Class	Method with Logging	.pex file
AuctionServerProject	Bid	CompleteBid	tool/aucesrv.pex
		StartBid	
	AuctionMgr	RequestAuctioneerStatus	
		AddBid	
		DeleteBidForPainting	
	RelinquishAuctioneerStatus		
ImageProject	ImageMgr	GetImage	tool/imageprj.pex

### *Logging Template*

The logging code used in the Auction application follows a coding template that tests for the log flags, sets up the parameters to output, invokes PutLine, then invokes Flush. The AuctionMgr.RequestAuctioneerStatus method is a good example:

#### **Code Example 6-4** Testing for log flags and writing log output

```
method AuctionMgr.RequestAuctioneerStatus(input password: string,
input name: string): integer
begin
    ...
    -- If debugging is on

    -- Test whether the trc:user1:1:1 log flag is set
    if task.Part.LogMgr.Test(
        type=SP_MT_DEBUG, serviceId=SP_ST_USER1, group=1, level=1) then

        -- Set up the string to output
        t : TextData = new;
        t.ReplaceParameters(
            'AuctionMgr.ReqAuctStatus: Name: <%1> Return status is <%2>.',
            TextData(value = name),
```

**Code Example 6-4** Testing for log flags and writing log output (*Continued*)

```

        IntegerData(value = retval));

    -- Use PutLine to log the string, then Flush the log buffer
    task.Part.LogMgr.PutLine(t);
    task.Part.LogMgr.Flush();
end if;
...
end method;

Proj: AuctionServerProject • Class: AuctionMgr • Method: RequestAuctioneerStatus

```

## Running the Example

To test your logging, you must set the log flags, run the application, then read the logged output.

► **To test logging instrumentation on the Auction application**

1. Set the current log flags settings to `trc:user1:1:1`.

For example, add this string to your `FORTE_LOGGER_SETUP` variable specification:

```
c:\forte\log\testlog.txt (trc:user1:1:1)
```

This will send the logging to a file in the `FORTE_ROOT` log directory on a Windows client.

Alternatively, you could set these flags using the iPlanet UDS Control Panel, or the Modify Log Flags dialog in the Repository Workshop. This would send logging output to stdout.

See [“Setting up Logging with iPlanet UDS” on page 196](#), for information on setting the log flags. See [“Locating Logging Output” on page 210](#) for information on the definition of stdout, given the method you use to run the application (local or distributed testing).

2. Run the application.

Run it either from the Repository Workshop or the Partition Workshop. See [“Choosing a Testing Mode” on page 209](#) for information on different methods for testing your application.

3. Read the output.

Depending on your platform, and how you ran the application, the output will be in a file or a client stdout window.

## Logging for Performance Example

This example illustrates a method for logging performance information on specific methods, such as those that access a database. In this example, methods to be observed time their transactions and then call a method to log the result to one of three possible output formats:

- normal output to a log
- comma-delimited output to a log
- a posted event with a message

Each method to be timed uses methods on the `Framework.StopWatch` class to record the elapsed time. These methods then pass the results to a log timer method to log the output.

Since every method set up to be logged runs a timer, whether or not the output method is triggered log the data, this technique involves a certain performance cost. Unless such a mechanism is part of the functionality of the application, this is the sort of logging you will want to remove once you have finished using it.

### Log Flags Used

This example sets up log flag constants for different functions and levels of detail. Any method can call the timer method on the condition the appropriate flags are set. The log flags used in this example are:

- message type (provided by iPlanet UDS; see [“Message Types” on page 185](#))
  - `SP_MT_PERFORMANCE`
- service types (provided by iPlanet UDS; see [“Service Types” on page 186](#))
  - `SP_ST_USER5`—for output to a log file in normal, human-readable format
  - `SP_ST_USER6`—for output to a log file in comma-delimited format (for reading into a spreadsheet)
  - `SP_ST_USER7`—for posting an event with a parameter containing the message
- group number and its corresponding value
  - `K_DB_GROUP = 1` (for database operations)

- level numbers and their corresponding values
  - K\_HIGH\_DETAIL = 255
  - K\_MED\_DETAIL = 125
  - K\_LOW\_DETAIL = 1

### Example Method That Times Itself

The following example method shows how the method times itself and passes the results to the method that outputs the information (LogTime).

#### Code Example 6-5 Method that times itself

```

method CustomerMgr.DeleteCustomer(
input custNo: Framework.IntegerNullable): Framework.integer
begin

  -- Delete a customer from the database
  try : integer - 0
  swatch : Stopwatch = new; // Create Stopwatch object
  methodName : string = 'DeleteCustomer';
  rows : integer;
  swatch.Fire(); // Start the timer

  retry : while ( try < K_MAX_RETRIES ) do //Start doing stuff
    begin transaction
      ...
    end transaction;
  end while; // Finish doing stuff

  // Stop the timer and
  // call LogTime to log the results
  self.LogTime( self.ClassName, methodName, swatch.split(),
               K_MED_DETAIL );
  methodName = '';
  swatch = NIL;

  return rows;
end method;

```

## Example Timer Output Method

The LogTime method called by the previous method is shown here.

### Code Example 6-6 LogTime method from Code Example 6-5

```

method My_Class.LogTime(input className: Framework.TextData,
input methodName: Framework.TextData,
input elapsedTime: Framework.integer,
input level: Framework.integer)

begin
-- Send performance information to the log
-- if the right log flags are set

logger : LogMgr = task.part.LogMgr;

//Logs to a normal log output if User5 is the service set in the
log flag setting.
if logger.test( SP_MT_PERFORMANCE, SP_ST_USER5, K_DB_GROUP,
               level ) then
    msg : TextData = new( value = '==> Class: %1,\tMethod: %2,
                               \tElapsed time: %3' );
    msg.replaceParameters( source = msg, parameter1 = className,
                          parameter2 = methodName,
                          parameter3 = IntegerData( value = elapsedTime ) );
    logger.putline( SP_MT_PERFORMANCE, SP_ST_USER5, K_DB_GROUP,
                  level, msg );
end if;

// Logs to a spreadsheet format if User6
// is the service type requested
if logger.test( SP_MT_PERFORMANCE, SP_ST_USER6, K_DB_GROUP,
               level ) then
    msg : TextData = new( value = '%1,%2,%3' );
    msg.replaceParameters( source = msg, parameter1 = className,
                          parameter2 = methodName,
                          parameter3 = IntegerData( value = elapsedTime ) );
    logger.putline( SP_MT_PERFORMANCE, SP_ST_USER6, K_DB_GROUP,
                  level, msg );
end if;

// Posts an event that logs to a screen message if User7
// is the service type specified in the log flags
if logger.test( SP_MT_PERFORMANCE, SP_ST_USER7, K_DB_GROUP,
               level ) then
    msg : TextData = new( value = '==> Class: %1,\tMethod: %2,
                               \tElapsed time: %3' );
    msg.replaceParameters( source = msg, parameter1 = className,
                          parameter2 = methodName,
                          parameter3 = IntegerData( value = elapsedTime ) );
    post self.elapsedTime( msg );

end if;
end method;

```



# Deployment Concepts

This chapter provides an overview of the information you need to deploy iPlanet UDS applications.

Included in this chapter is conceptual information about the following topics:

- deploying applications and libraries
- configuring applications
- configuring libraries
- environments

**Chapter 8, “Deploying iPlanet UDS Applications and Libraries,”** leads you step by step through the process for deploying an application and a library.

## An Overview of Deploying Applications and Libraries

This chapter explains some concepts and parts of iPlanet UDS that you need to be familiar with to be able to deploy iPlanet UDS applications and libraries (which are a special case of applications).

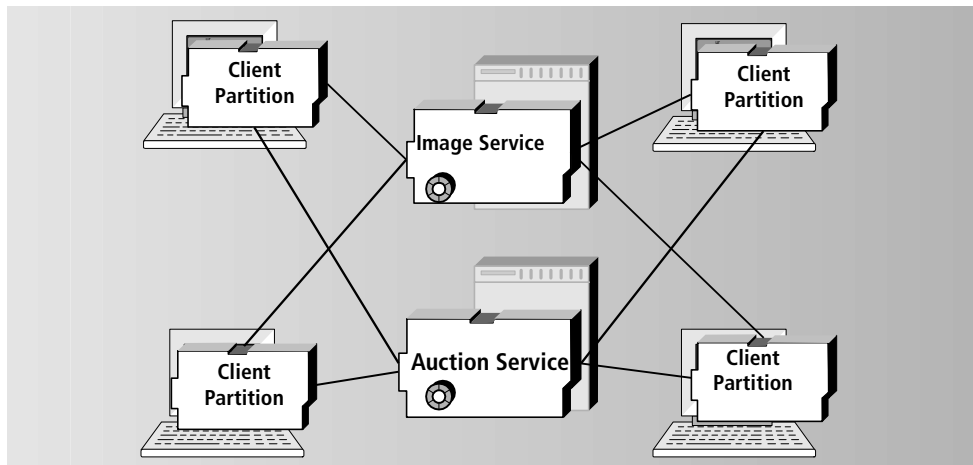
When you develop a TOOL application, you can write many parts of your application without considering exactly what the deployed configuration of your application will be or on what physical network and machines your application runs.

## Distributed Applications and Application Distributions

iPlanet UDS applications are typically *distributed applications*, which means that different parts of the application run on different machines in the environment. When you configure the application, you define the logical sections, called *partitions*. Each partition is an independent process that can run on its own machine. When you configure the application, you also define on what machines these partitions will run. You can then create an *application distribution*, which contains the application files needed to run the application on all the defined machines. Finally, you can install the application in any environment that contains machines like those specified in the configuration.

The server partitions run on server nodes, which might be serving one client or any number of clients. The client partitions run on client nodes, which might be communicating with any number of servers. **Figure 7-1** illustrates the partitions in a distributed application.

**Figure 7-1** Partitions in a Distributed Application



**Chapter 8, “Deploying iPlanet UDS Applications and Libraries,”** describes the detailed steps for configuring and deploying applications.



## Libraries

A *library* is a named collection of classes and other component definitions that can be shared by any number of iPlanet UDS applications. iPlanet UDS provides several libraries for your use in developing applications, including Framework, GenericDBMS, and Display. You can also create your own libraries for use within your organization or for distribution or sale to other companies. For example, you could create a library that provides access to an external service, such as a stock market service, or a collection of statistical routines.

When the library is installed in a development environment, the library can be imported into any of the development repositories in the environment. Then, any projects being developed in those repositories can include the library as a supplier plan. The ability to include libraries as supplier plans allows multiple projects within different development repositories to share a single library.

“[Deploying a Library](#)” on page 304 describes in detail how to configure and deploy a library.

## Environments

When you prepare to deploy your application, you need to define a configuration and map your application configuration to the definition of a network. In iPlanet UDS, the definition of a physical network where you can install and run iPlanet UDS applications is called an *environment definition*, which is described in “[Configuring Applications](#)” on page 225.

# Configuring Applications

To distribute an application, iPlanet UDS divides the application into logical sections, called partitions. Each partition is an independent process that can run on its own machine. You build a *configuration* to customize your application for the particular hardware and software in that environment by assigning the partitions to nodes in the environment. A configuration also allows you to replicate certain partitions for load balancing or failover. The result is a distributed application from which you can generate the files to be installed in the environment.

There are two kinds of application configurations: a client configuration and a server configuration

## Client Configuration

A *client configuration* defines a client application, which is an application that the end user can run from his or her workstation. A client configuration contains one client partition, which includes the user interface for the application (if there is one) and the startup code. A client configuration can also contain one or more server partitions.

### Applets

Using the Partition Workshop, you can configure any client configuration as an applet, rather than as a client application. An *applet* is an application that can be launched from another application using the AppletSupport library, but cannot be run as an independent client application. Except for this limitation in the way it can be started, an applet is the same in every way as a client application. For complete information on applets, see [“Writing Applications That Use the Launch Server and Applets” on page 435](#).

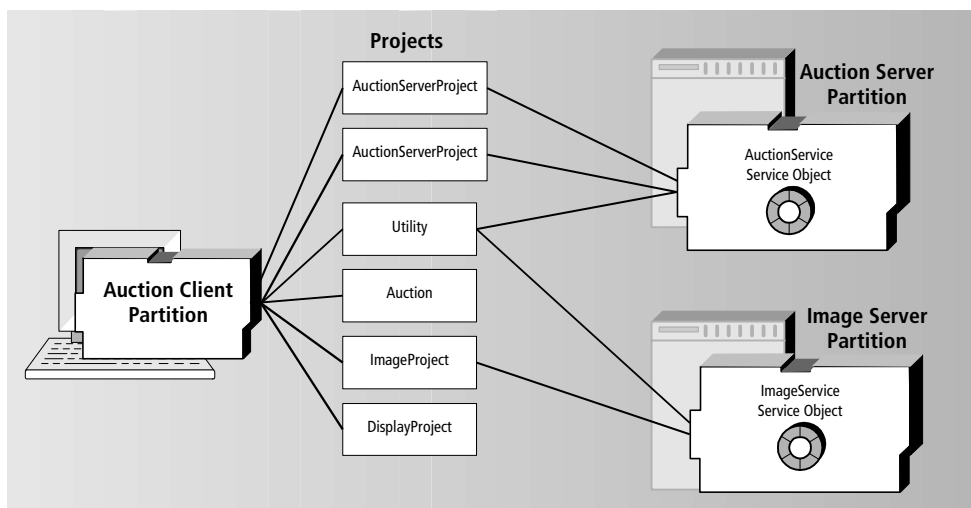
## Server Configuration

A *server configuration* defines an iPlanet UDS server application, which provides processing for one or more client applications. A server configuration has one or more server partitions, and no client partition.

## Relationship Between Partitions and Projects

An application consists of a main project and all of its supplier plans. When iPlanet UDS partitions your application, it assigns all of the service objects in the main project and its supplier projects to partitions. If any class within a project is needed by a service object on a partition, iPlanet UDS also includes that project definition on the partition. Therefore, a single project may appear on any number of partitions.

For example, in the Art Auction application, the Painting class needs to be available on the client partition because it is displayed to the end user on the window; it also needs to be available on the server partition where the Image Server is running because that is where the paintings are stored. Therefore, the project that defines the Painting class needs to be on both partitions. [Figure 7-2](#) illustrates the relationship between partitions, service objects, and projects:

**Figure 7-2** Relationship between Partitions and Projects

## Relationship Between Partitions and Libraries

The supplier libraries for the application must also be installed on the partitions that need to access them. However, the application configuration does not include the supplier libraries (only the supplier projects). Therefore, you must make a separate library distribution that contains the libraries needed by the application, and install the library distribution along with the application distribution. See [“Configuring Libraries” on page 233](#) for information.

## Automatic Partitioning

iPlanet UDS automatically partitions your application for each of your environment definitions. Most of your service objects can run only on a certain node. Therefore, iPlanet UDS automatically assigns the service objects to specific partitions and assigns the partitions to the nodes in the environment definition that have the appropriate resources and capabilities. For further details on automatic partitioning, see [“Creating a Default Application Configuration” on page 240](#).

After iPlanet UDS partitions the application, you can examine it and make adjustments. If necessary, you can assign service objects to different partitions or assign partitions to different nodes. For partitions that contain replicated service objects, you can replicate the partitions to provide load balancing or failover.

When you examine a partition in the Partition Workshop, you see only the service objects assigned to the partition, not the projects. However, iPlanet UDS ensures that the project definitions needed by each service object are always included on the partition. If you move a service object, the projects it needs are moved along with it.

(Note that you can use the **ShowApp** command in Fscript to see which projects are included on a partition. See the *Fscript Reference Manual* for information on the **ShowApp** command.)

## Testing a Configuration

After you adjust your configuration, you can test it by running the partitioned application in the development environment. This lets you test the distributed application as it will appear to end users when it is installed.

## Deploying Applications

The completed configuration is your “distributed application.” When your configuration is ready, you can generate the *distribution*, that is, the files necessary to install the application for production deployment. The Partition Workshop provides a Make Distribution command for this purpose. See “[Making an Application Distribution](#)” on page 269 for information.

## About Partitions

A partition is a logical section of an application that represents either the client portion of the application or one of the servers. When iPlanet UDS partitions an application, it assigns all the service objects in the application to partitions; these are the *logical partitions*. iPlanet UDS then assigns the logical partitions to appropriate nodes in the environment definition; these are the *assigned partitions*.

The Partition Workshop shows both the logical partitions into which your application is divided and the assigned partitions that represent the processes that will be installed in the environment. The following sections provide information about the properties of logical and assigned partitions.

## Logical Partitions

When iPlanet UDS partitions your application, it divides the application into one client partition and any number of server partitions.

### Client Partition

A client partition contains the user interface for the application (if there is one) and the application startup code. In addition, if the application contains any service objects with user visibility, iPlanet UDS will put them on the client partition if the client nodes in the environment can support them. If you used the `Configure as > Server` command to create a server configuration, iPlanet UDS does not create a client partition for the application.

### Server Partition

A server partition contains one or more service objects and the projects accessed by the service objects. There are two types of server partitions: replicated and non-replicated.

### Replicated Server Partition

A replicated server partition is a partition that contains a service object that was defined as being replicated for load balancing or failover. When a logical partition is replicated, you can assign it to any number of nodes in the environment. For each assigned partition, you can specify a startup replicate count, as described under [“Assigned Server Partition Properties” on page 266](#).

### Router Partition

When the service object in a partition is replicated for load balancing, iPlanet UDS automatically creates an extra partition called a router partition. The purpose of a router partition is to route the traffic between the partitions that are load balancing work for the service object. Although the router partition is usually assigned to the same node as one of the server partitions that it is managing, it can be on any server node in the environment.

If the service object is replicated for both failover and load balancing, iPlanet UDS replicates the router partition for failover and replicates the server partitions for load balancing. As a result, if the first router partition fails, the second router partition can take over and manage the partitions that are sharing the work load, and so on for each replicated router partition. See [“The Router Partition” on page 424](#) further information about the router.

Although the router partition appears in the Partition Workshop as a standard partition, we strongly recommend that you do not move any service objects onto this partition.

## Non-replicated Server Partition

A non-replicated server partition is a partition containing service objects that were not defined as being replicated. When a logical partition is non-replicated, you can assign only one enabled partition in the environment. Other copies you make of the non-replicated server partition will be automatically disabled, as described under [“Assigned Server Partition Properties” on page 266](#).

In the Partition Workshop, you can change whether or not a partition is replicated by modifying the definition of the service object that it contains. See [“Modifying a Service Object Definition” on page 260](#) for information.

## Reference Partition

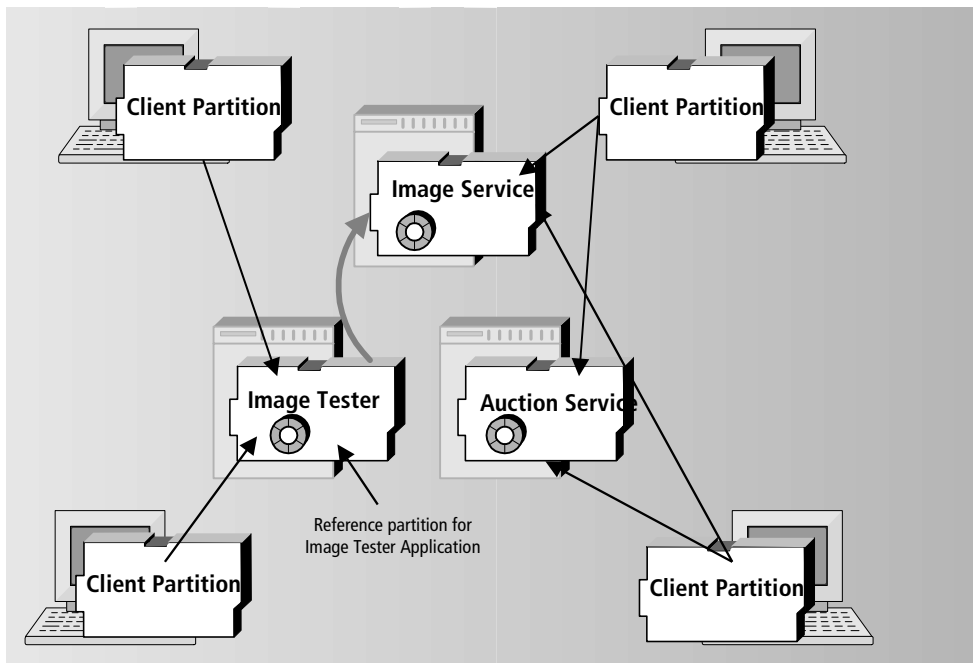
A special kind of logical partition is a *reference partition*. A reference partition allows you to share an existing service object in another application (or in another environment) rather than starting a new instance of the server partition in the current application.

To share a service object between applications, you first deploy the service object in one application. The application that deploys the service object can be either a server application or a client application—it makes no difference.

Then, in the other applications that need to access that deployed service object, you create a *reference partition*. Instead of containing a new service object, the reference partition points to the existing service object that was originally deployed as part of the first application.

### *Sharing Services Objects in the Current Environment*

Reference partitions allow you to create business services that are shared by any number of applications in the current environment. For example, if you want your application to interact with an image server that is already provided by another application in the environment, you can include the project that defines that service object as a supplier and then create a reference partition that points to the existing service object. This way your application will be sharing the existing service object with any other applications that are using it, rather than creating a new instance of the service object. [Figure 7-3](#) illustrates a reference partition:

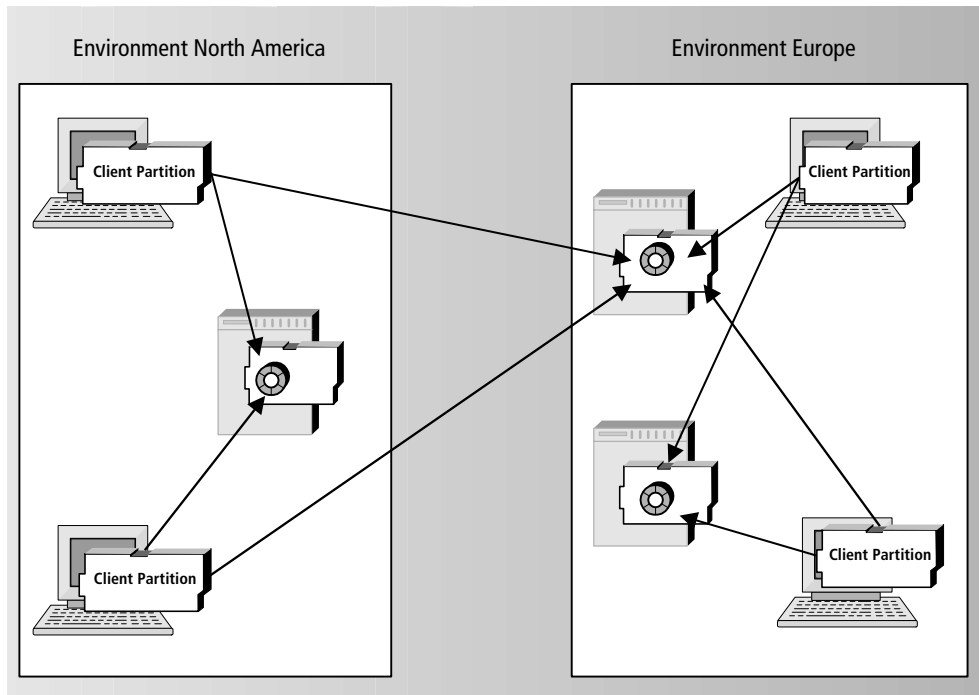
**Figure 7-3** Reference Partition for Current Environment

### *Sharing Services Objects Between Connected Environments*

If your deployment environments are connected, you can create business services that are shared across environments. For example, you may have a service object that can run only in one environment, such as a service object that uses a specialized satellite feed or stock ticker that can be accessed through a callout to an external service running in one location. Or, imagine you have a service object that handles personnel data. This service object only resides in the headquarters environment because that is where the personnel database is. All other applications installed in their own local environments must access the personnel database located at headquarters.

Applications in other environments can access the specialized service object in the connected environment by using a reference partition. The following figure illustrates:

**Figure 7-4** Reference Partition for Connected Environment



The advantages of using reference partitions include:

- modularity

The ability to share a single service object between multiple applications means that you need to create and manage only that single service object. Without reference partitions, you would need to create more than one version of the service object.

- efficient use of resources

Only one service object needs to be running to provide services needed by multiple applications. Without reference partitions, you would need to run the service object within every application that needs its services.

For complete information about reference partitions, see [“Making a Reference Partition”](#) on page 251.



# Configuring Libraries

A library can be installed in both deployment environments and development environments. When the library is installed in a deployment environment, any number of applications running in the environment can reference it. This provides the ability for multiple applications to share a single library, which makes installation easier and more efficient.

A *library distribution* is a set of related libraries packaged together for distribution. This provides a convenient way to bundle any number of related libraries together for deployment and installation. When you bundle libraries into a library distribution, you can make a single library distribution to use for deployment.

To create a library, you must define a project that contains the definitions you want to be shared and then configure that project as a library. A project becomes a “library” rather than a project when you include that project in a library distribution. There is a one-to-one correspondence between the *projects* you include in a library distribution and the *libraries* that are available when that library distribution is deployed. Each project in the library distribution gets compiled into a separate shared library.

For assigned libraries, you can specify whether the library on a given node is standard or compiled. The option of providing compiled libraries provides improved performance. A *compiled library* uses C++ code generation to create a compiled shared library. Normally, the libraries running on server machines are compiled. However, by default, all libraries are standard because installation and management of standard libraries is simpler.

The ability to share a library rather than duplicating its definitions provides efficiency gains. You can improve performance further by providing compiled libraries. A *compiled library* uses C++ code generation to create a compiled shared library. Normally, the libraries running on server machines are compiled. However, by default, all libraries are standard because installation and management of standard libraries is simpler.

If a given library has supplier libraries, you must be sure to set the compilation options correctly for the main library and its suppliers. Once you designate a given library on a particular node as compiled, you must ensure that all its supplier libraries are also compiled. For a standard library, the supplier libraries can either be standard or compiled.

When deciding whether or not a library is compiled, you also need to take into account whether or not the partition that will be using the library is compiled. If the partition is compiled, the library that it accesses must also be compiled. If the partition is standard, the library that it access can be either standard or compiled.

A library distribution has a compatibility level based on the compatibility level of the project for which you give the Configure as > Library command. Like an application's compatibility level, the library distribution's compatibility level allows you to release different versions of the same library within a single environment. The rules for when you need to raise the compatibility level of a library distribution are the same as those for an application. See [Chapter 13, "Upgrading Deployed Applications"](#) for information.

## About Environments

An environment is the distributed system on which you run a distributed iPlanet UDS application. An iPlanet UDS environment may consist of one or hundreds of computers. A local- or wide-area network might contain several iPlanet UDS environments. An iPlanet UDS developer or system manager uses the Environment Console to define an environment by describing a set of nodes, each of which corresponds to a particular machine in the environment. Only after an environment has been defined can you partition the application to run in that environment. When you partition the application, you can select the environment definition you want to use.

There are two kinds of environments in iPlanet UDS: development environments and deployment environments.

### Development Environment

A development environment is an environment in which the development version of iPlanet UDS is installed. You create and test all your iPlanet UDS applications in the development environment. In the Partition Workshop, you can partition your application in the development environment for initial testing of the distributed version of the application.

### Deployment Environment

A deployment environment is an environment in which the runtime version of iPlanet UDS is installed. You can have any number of environment definitions that represent the deployment environments in which you plan to install the production version of your application. Each deployment environment can consist of a different number of nodes, with varying architectures. In the Partition Workshop, you partition your application for each of the deployment

environments for final testing. When your application is complete, you use the Partition Workshop to create an application distribution for each of the deployment environments. An application distribution contains the files necessary for actually deploying the applications.

## Simulating Deployment Environments

Because you cannot run the Partition Workshop in the deployment environments (and you may not even have physical access to them), iPlanet UDS simulates the deployment environments by using nodes in your development environment as stand-ins for nodes in the deployment environment. Every node in an environment definition is assigned to a “testing node,” which is the node in the development environment that is used to simulate the deployment node for testing. (Of course, to simulate your deployment environments, your development environment must include at least one of each machine type that will be included in all your deployment environments.)

In the Partition Workshop, you select the environment into which you wish to partition your application. This environment can either be the development environment or a deployment environment; a deployment environment is simulated within the development environment. While you are working in the Partition Workshop, you can examine the node definitions for the environment in which you are partitioning your application, as described in [“Examining Nodes in an Application Configuration” on page 245](#). However, you cannot change the node definitions or any other part of the environment definition from the Partition Workshop. The Environment Console or the Escript utility, its command-line counterpart, are the only utilities that allow creation and modification of environment definitions.

## Connected Environments

iPlanet UDS environments can be connected, that is, partitions in one environment can find service objects in other iPlanet UDS environments and access them. See [“Using Reference Partitions with Connected Environments” on page 254](#) and [“Specifying the Environment Search Path” on page 262](#) for information about using service objects from other environments.

## Nodes

A node is a machine in your network that is capable of running an iPlanet UDS partition. In the Partition Workshop and Fscript, you can examine the nodes in the environment by viewing the node's properties dialog. You cannot, however, change node definitions from the Partition Workshop or Fscript.

Every node has the following properties:

<b>Node Property</b>	<b>Description</b>
Name	The name assigned to the node when the environment was defined.
Architecture	The node type.
Testing Node	The name of the node in the development environment used to simulate the current node for testing.
Client	Specifies that you plan to install client partitions on the current node. A node where you plan to install only server partitions is considered a server node.
Use as Model	Specifies that the current node represents a number of identical client nodes.
Use for Testing	Specifies that the node can be used for simulating a node in a deployment environment.
Resource Managers	The list of resource managers available on the current node.
Installed Protocols	The list of communication protocols that the node can use to communicate with other iPlanet UDS nodes.
Installed Libraries	The list of restricted, user-defined external projects installed on the current node.

For more information about nodes and environments, see *A Guide to the iPlanet UDS Workshops* and the *iPlanet UDS System Management Guide*.

# Deploying iPlanet UDS Applications and Libraries

After you develop your application or library, you need to configure the application or library and deploy it onto the network.

This chapter describes how to deploy your applications and libraries, and includes the following information:

- an overview of the various ways you can configure and deploy an application
- how to configure applications and libraries
- how to package libraries and applications by making a distribution
- how to install applications and libraries in your environment

## About Deploying Applications and Libraries

When you develop iPlanet UDS applications and libraries, you first need to design and write the necessary TOOL code. Finally, after you have thoroughly tested the application or library, you need to perform the following tasks to provide the application to users, as illustrated in [Figure 8-1](#):

1. Define a configuration (also called partitioning).

When you define an application configuration, you make decisions about what parts of your application are going to run together as a partition, and on what kinds of machines. You can also specify settings, such as whether the partition is compiled.

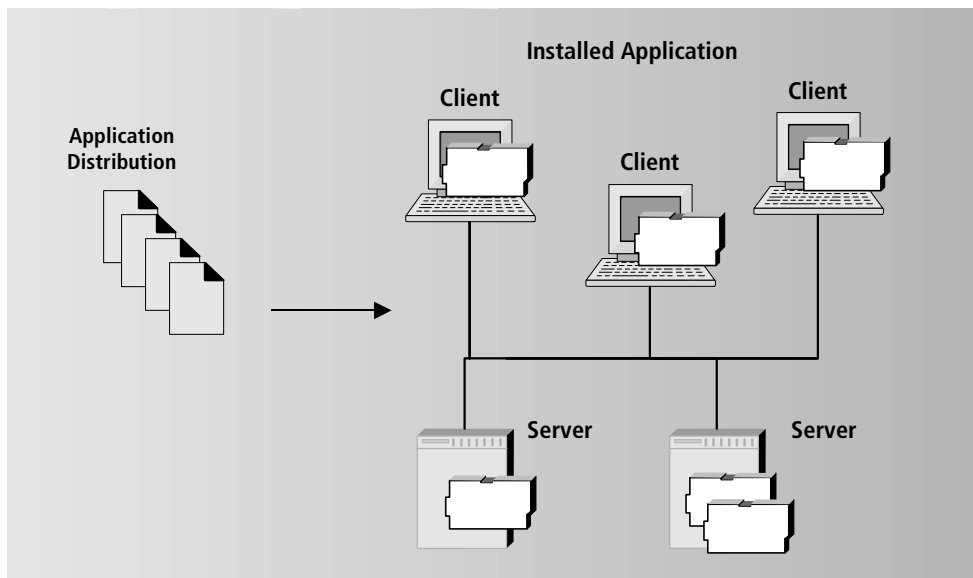
2. Make a distribution.

The files that make up the installable iPlanet UDS application or library are collectively called the *distribution* for that application or library. You can then move and copy a distribution to distribute and install the library or application.

3. Install the application or library.

iPlanet UDS provides system management tools that let you install and manage applications and libraries on a network that is defined as an iPlanet UDS environment.

**Figure 8-1** Deploying an Application or Library



## iPlanet UDS Utilities for Deploying Applications and Libraries

iPlanet UDS provides the following utilities that enable you to deploy iPlanet UDS applications and libraries:

**Partition Workshop** An iPlanet UDS workshop that is available only when you run the iPlanet UDS workshops in distributed mode. You can define an application or library configuration in the Partition Workshop. You can also make a distribution and install the application or library in the current development environment. The Partition Workshop is described in detail in *A Guide to the iPlanet UDS Workshops*.

**Fscript** A command-line utility that lets you interact with the development repositories and the iPlanet UDS configuration features. You can write scripts to automate these commands. Fscript is described in the *Fscript Reference Manual*.

**Environment Console** An application that graphically represents the parts of an iPlanet UDS environment. You can use the Environment Console to install, monitor, and manage applications in the environment. The Environment Console is described in the *iPlanet UDS System Management Guide*.

**Escript** A command-line utility that lets you interact with the iPlanet UDS system management features. You can write scripts to automate these commands. Escript is described in the *Escript and System Agent Reference Manual*.

## Examples

The example used in this chapter is the Auction example, which is described in [“Auction” on page 634](#).

## Getting Started

This chapter provides a step-by-step description of how to define a simple application configuration using the iPlanet UDS workshops and the Environment Console. For information about how to use Fscript and Escript commands to perform the same tasks, see the *Escript and System Agent Reference Manual* and the *Fscript Reference Manual*.

Before you can configure or deploy an application, you must be running the iPlanet UDS Workshops in distributed mode, which means you have clicked the iPlanet UDS Distributed icon or used the **forte** command without specifying the **-fs** flag. For more information about how to start the iPlanet UDS Workshops in distributed mode, see *A Guide to the iPlanet UDS Workshops*.

To deploy an application, you need to have write access to the volumes where you want to place the application distribution and where you want to install the application.

## Creating a Default Application Configuration

At this point, you should have written and test run your application in the iPlanet UDS Workshops or Fscript. You now need to define a configuration for your application so that you can install it in a deployment environment for testing and, eventually, customer use.

To create an application configuration, you need to define to which partitions different parts of the application belong. This task is described in [“Configuring Applications” on page 225](#), and is also referred to as *partitioning*.

The simplest way to define a configuration is to let iPlanet UDS create a default configuration, as described in the next section. However, you will usually want to customize your application configuration, as described in [“Customizing the Application Configuration” on page 247](#).

You can configure applications that contains both client and server partitions (client applications) or applications that contain only server partitions (server applications).

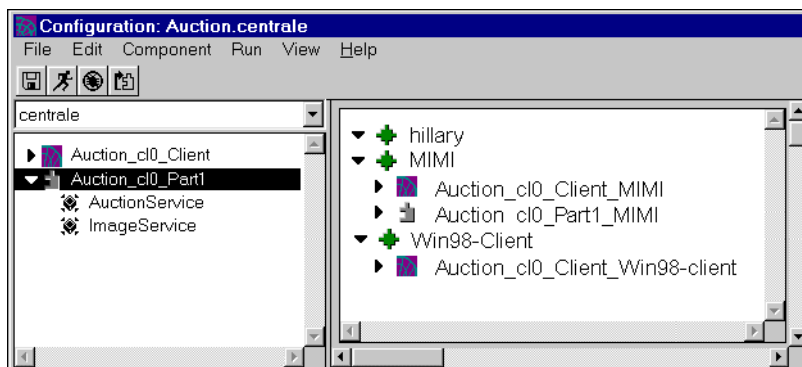


► **To create a default application configuration for client applications**

1. In the Project Workshop for the main project of the application, choose File > Configure as > Client.

The Partition Workshop opens, showing the default logical partitions for the application on the left and the default assignments of the partitions onto the nodes in the development environment.

The following figure shows a default configuration for the Auction client application.

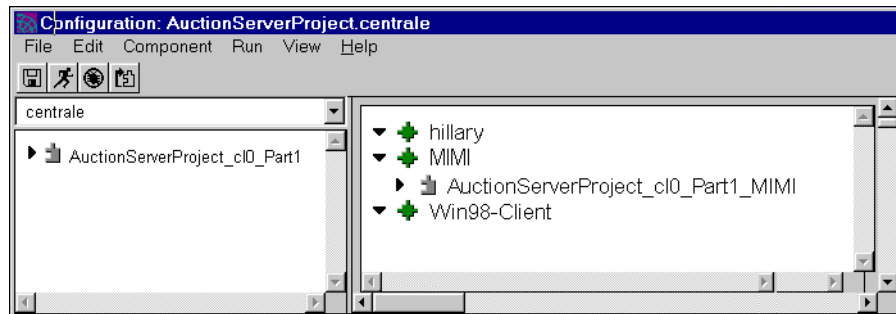


► **To create a default application configuration for server applications**

1. In the Project Workshop for the main project of the application, choose File > Configure as > Server.

The Partition Workshop opens, showing the default logical partitions for the application on the left and the default assignments of the partitions onto the nodes in the development environment.

The following figure shows a default configuration for the Auction server application.



In the above figures, the nodes of the environment definitions are:

**hillary** a server node running AIX

**MIMI** a client and server node running Windows NT

You can choose to use the default configuration, but you probably want to customize this configuration.

If you have already defined a custom configuration, but want to have iPlanet UDS replace that configuration with the default configuration, use the File > Repartition command.

## How the Default Configuration is Generated

When iPlanet UDS partitions an application, it assigns all the service objects in the main project and all its supplier projects to logical partitions.

### How Client Partitions Are Assigned

If the application has a client (that is, if you used the Configure as > Client command to create the configuration), iPlanet UDS creates a client partition to contain the user interface for the application and the application startup code. In addition, if the application contains any user-visible service objects, iPlanet UDS will put them on the client partition if the client nodes in the environment can support them. The client partition is then assigned to every client node in the environment.

## How Server Partitions Are Assigned

If a service object has environment visibility, iPlanet UDS assigns it to a server partition. If a service object has user visibility and cannot run on the client partition, iPlanet UDS assigns it to a different server partition than the one that contains the environment-visible service objects. Each server partition is then assigned to an appropriate node in the environment. If the partition requires a database resource manager or a restricted C project, iPlanet UDS assigns the partition to the node where the required resource or library is installed. If you have previously specified a default node and/or excluded nodes for the configuration (described in [“Changing Configuration Properties” on page 268](#)), this will be taken into account when the application is partitioned.

By default, iPlanet UDS assigns all compatible service objects to the same partition. However, when you are ready to deploy the application, you may need to move some of the service objects to partitions on different nodes in the deployment environment. You should do this before making the application distribution for the deployment environment. See [“Customizing the Application Configuration” on page 247](#) for information on how to make new partitions and move service objects.

## Why Some Service Objects Are Unassigned

If you have any service objects in the application that cannot be supported in the current environment (for example, because a DBMS resource manager service object is defined for an external manager that is not present in the environment, or because a restricted 3GL project has not been installed in the environment), these will be unassigned. To run your application in this environment, you must either update the service object definition or the system manager must update the environment definition so the service object can be assigned to a partition. You can then assign the partition to the appropriate node.

## Why Service Objects Are Not Replicated

iPlanet UDS does not automatically replicate your service objects for load balancing or failover. Instead, in the Partition Workshop, you must replicate the partitions to which the service objects are assigned. You can replicate a partition either by assigning it to an additional node (as described under [“Adding Partition Assignments” on page 264](#)) or by setting the replication count for an individual assigned partition on a single node (described under [“Changing Assigned Partition Properties” on page 265](#)).

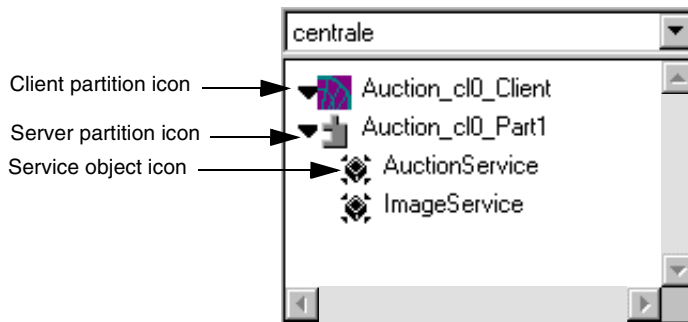
For each service object that is replicated for load balancing, iPlanet UDS automatically creates a router partition. The router partitions are assigned to the default server node for the environment.

## Examining the Logical Partitions

The Logical Partition browser portion of the Partition Workshop, shown in [Figure 8-2](#), displays a two-level hierarchy that lists the names of the logical partitions for the environment and the service objects contained in each partition.

For logical partitions, the browser displays the logical partition names. Icons indicate whether the partition is a client partition or a server partition. You can identify a router partition by its name.

**Figure 8-2** Logical Partition Browser



To view a list of the nodes to which the partition is assigned, double-click on the partition name. A dialog appears, as shown in [Figure 8-3](#), which lists the nodes where the partition is assigned and indicates whether the partition is enabled or disabled.

**Figure 8-3** Logical Partitions Dialog



For service objects, the Logical Partition browser displays the service objects' names. To examine the service object definition, double-click on the service object name. The Service Object Properties dialog opens, displaying the full definition of the service object.

If the service object is in a *reference partition*, the Service Object Properties dialog does not display the full definition of the service object. Instead, the dialog displays the name of the referenced application and referenced partition. (If the service object in the reference partition is in another environment, the Service Object Properties dialog shows the environment search path.)

To display the full definition of the service object, you must examine the configuration for the project where the service object was originally defined.

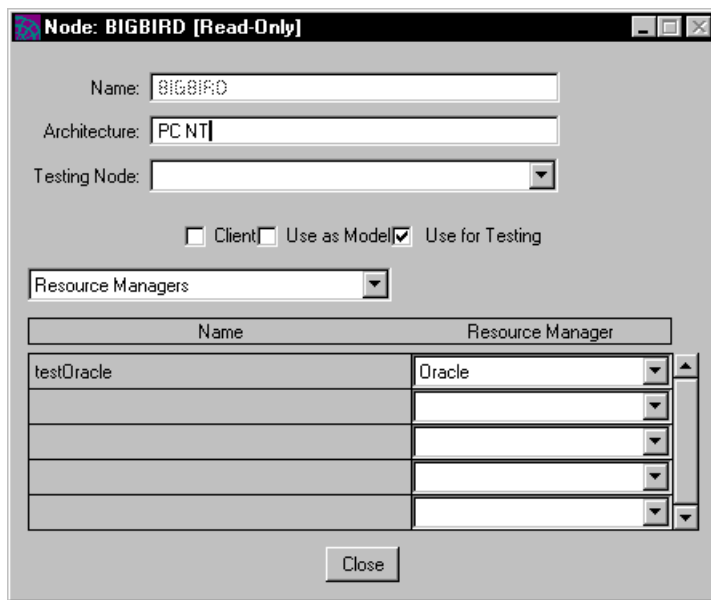
If there is a mismatch between the service objects defined in the application and the external managers or installed C projects provided by the environment, iPlanet UDS will not be able to assign all the service objects to logical partitions. In this case, the unassigned service objects will be displayed on the "Unassigned" list. Because you cannot run the application with unassigned service objects, you must fix this problem either by modifying the service object definition or having the system manager modify the environment definition. See ["Modifying a Service Object Definition" on page 260](#) for information on modifying the service object definition.

## Examining Nodes in an Application Configuration

The Nodes browser displays the assigned partitions within the environment. By default, the browser displays a topological view of the environment. The View > Node Outline command lets you display the same information in an outline form.

For each node in the environment, the browser displays a three-level hierarchy. At the top level is the node name. At the second level are the partitions assigned on that node. At the third level are the service objects assigned to each partition.

To examine the node properties, double-click the node name, or select the node and choose the Component > Properties... command. The Node dialog appears, shown in [Figure 8-4](#), displaying the node properties described under ["About Environments" on page 234](#). These properties are the settings that were specified for the node when the environment was created in the Environment Console, and you cannot change them from the Partition Workshop. See the *iPlanet UDS System Management Guide* for further information on the properties of a node.

**Figure 8-4** Node Properties Dialog

## Examining the Assigned Partitions

For each assigned partition, you can examine information about the partition as well as information about any DBSession service objects on the partition. To open the Assigned Partition Properties dialog, double-click the assigned partition name or select the partition and choose the Component > Properties... command. See [“Changing Configuration Properties” on page 268](#) for information about the properties available for assigned client partitions and for assigned server partitions.

You cannot use the Partition Workshop to see which projects in your application are assigned to a particular partition. However, you can use Fscript to do so. The ShowApp command in Fscript shows the current configuration information for the current application in the current environment. See the *Fscript Reference Manual* for information on the ShowApp command.

# Customizing the Application Configuration

You can customize application configurations in any of the following ways:

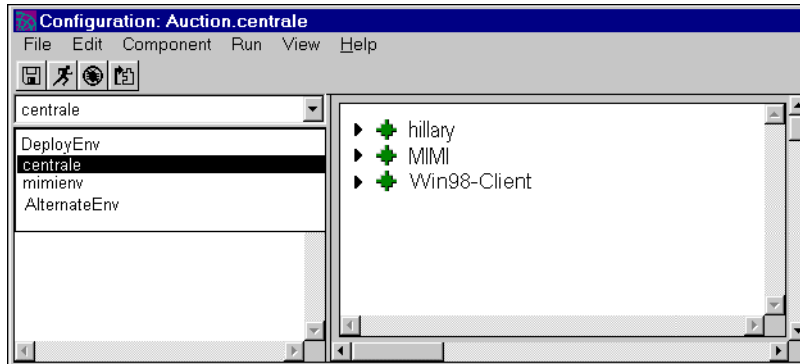
- choose a simulated environment definition, as described in [“Choosing a Simulated Environment Definition” on page 249](#)
- redefine logical partitions, as described in [“Redefining Logical Partitions” on page 249](#), which affects all assigned copies of that partition
- modify a service object definition, as described in [“Modifying a Service Object Definition” on page 260](#)
- change partition assignments and add partition assignments, as described in [“Changing Partition Assignments” on page 264](#)
- change assigned partition properties, as described in [“Changing Assigned Partition Properties” on page 265](#)
- change configuration properties, as described in [“Changing Configuration Properties” on page 268](#), which causes iPlanet UDS to repartition the application

Note that the application configuration does not include the libraries that are needed by the application. If your application has supplier libraries, you must create a separate library configuration for the libraries, as described under [“Modifying a Library Configuration” on page 308](#), and then create a separate library distribution for them as described under [“Making a Library Distribution” on page 312](#). Both the application distribution and library distribution must be installed for the application to run.

To modify a configuration, you must first open it.

► **To open a configuration**

1. In the Project Workshop for the main project, choose File > Configure as > Client or Server, depending on the type of configuration.
2. After the Partition Workshop opens, choose the name of the environment you want to use, as shown in the following figure:



See “[About Environments](#)” on page 234 for a description of the relationship between a configuration and an environment.

If you select an environment that does not yet have a configuration, iPlanet UDS automatically partitions the application. If you select an environment that already has a configuration and you have changed any of your projects in such a way as to affect the configuration, iPlanet UDS automatically performs an incremental partitioning on the application, however, this may take some time.

Note that to make permanent changes to a configuration, your workspace must be open for updating. If your workspace is open for reading only, you can use the Partition Workshop to examine a configuration, test it, and make temporary modifications to it. However, because you cannot save your workspace, any changes you make to the configuration are only temporary.

You test a client configuration in the Partition Workshop by selecting an environment in which to run the client application. For information about testing a client configuration in the Partition Workshop, see *A Guide to the iPlanet UDS Workshops*.

Debugging the application from the Partition Workshop allows you to run the application using the current configuration and to step through the code running on the client partition. You cannot use the iPlanet UDS Debugger to monitor code running on remote partitions. For information about debugging a client partition in the Partition Workshop, see *A Guide to the iPlanet UDS Workshops*.

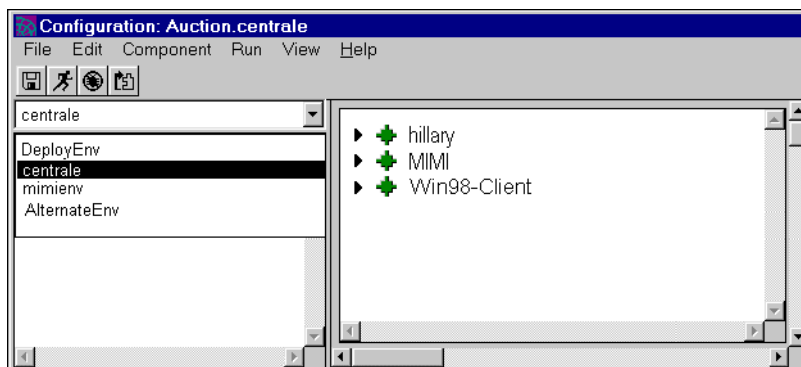


## Choosing a Simulated Environment Definition

By default, iPlanet UDS creates the default configuration based on the definition of the current development environment.

You can choose to create a configuration for another environment definition by choosing one of the names in the environment drop list, as shown in [Figure 8-5](#). Note that if any environment definitions are currently locked by the Environment Console or the Escript utility, you cannot use them.

**Figure 8-5** Choosing the Name of a Simulated Environment



iPlanet UDS creates a configuration for an application in a particular environment. Therefore, you can define one configuration for each environment definition shown in the environment drop list.

## Redefining Logical Partitions

You can make the following changes to a logical partition:

- move a service object from one partition to another
- create a new logical partition to contain the selected service object
- make a reference partition
- make an applet
- modify the definition of an individual service object

## Moving Service Objects Between Partitions

You can move a service object to any compatible partition. For the partition to be compatible, the service objects in the target partition must meet the following conditions:

- If the service object you want to move is replicated, the service objects in the target partition must be replicated the same way.
- If the service object you want to move is associated with a resource manager, the service objects in the target partition must be associated with the same resource manager.
- If the service object you want to move belongs to a restricted project, the service objects in the target partition can belong to another restricted project only if both restricted projects can run on the same node.

If you do try to move a service object to an incompatible partition, you will get an error message that explains why it is incompatible.

You can include a user-visible service object in the same partition as an environment-visible service object. However, when a user-visible service object is in the same partition with an environment-visible service object, the user-visible service object becomes accessible only within that partition. This limited accessibility can be a useful optimization technique. See [“Combining Service Objects and Partitions” on page 257](#) for more information.

### ➤ To move a service object

1. In the Logical Partition browser, select the service object you wish to move.
2. Drag the service object on top of the target partition name.

If the partition is incompatible, iPlanet UDS will display an error dialog explaining why the service object cannot be moved.

## Creating a New Logical Partition

To create a new logical partition, you must have at least one existing service object that you wish to assign to it.

### ➤ To create a logical partition

1. Select the service object you wish to move to the new partition.
2. Choose the Component > New Logical Partition command.

A new logical partition containing the service object appears in the list of logical partitions.

## Making a Reference Partition

Before you can make a reference partition, the application that includes the service object must be deployed. Then, you must obtain the project that defines the service object and include it as a supplier plan for your main project.

The project that you include as a supplier plan for your main project must be the *same* project that was used to define the service object in the deployed application. The supplier project in the current application must match the project in the deployed application in the following ways:

- the names of the projects must be the same
- the compatibility levels of the projects must be the same
- all runtime IDs must match

---

**CAUTION** If the project that defines the service object was created in another repository, you need to export the project from the other repository using the Fscript **ExportPlan** with the **ids** option. Then, you need to import this exported plan into your repository.

---

Any time that multiple repositories are using the same project, the project must have been imported with IDs into all the repositories (except for an originating repository) even when the project was originally imported into a repository without IDs. Whenever a project without IDs is imported into a repository, new IDs are automatically created for the project, its components, and its service objects. If you simply import the same exported project without IDs into different repositories, applications generated by these separate repositories will not recognize that the projects, service objects, and components are the same because the IDs are different.

If you have received an exported project that does not include unique IDs, you need to get another copy of the exported project that includes these IDs; otherwise, your application will not be able to use the deployed partition. The iPlanet UDS runtime system uses the IDs to determine whether the requested service object and the available service objects are the same.

You must perform the following steps to include a service object from a deployed application in your current application.

➤ **To include the project that defines the service object**

1. Deploy the application that defines the service object that is to be shared.
2. If the service object is in a connected environment, you must make a distribution in the current environment for the server application that contains it (although you do not install it).
3. In the application that needs to access the deployed service object, include as a *supplier plan* the project from the deployed application that originally defined the service object.
4. Use the Partition Workshop to create a reference partition that points to the existing service object.

If the service object is in a connected environment, the environment search path for the service object in the reference partition must specify the environment where the original server application that contains the service object is deployed.

The remainder of this section describes how to use the Partition Workshop to create a reference partition. See [“Sharing Service Objects Between Applications” on page 426](#) for detailed information on [Step 1](#) through [Step 3](#).

When you partition your application in the Partition Workshop, iPlanet UDS creates new partitions containing all the service objects in the main project and all of its supplier projects, just as usual. Therefore, the service object that you are planning to include in the reference partition will be on a new partition (or it may be sharing a new partition with other service objects).

At this point, you must create the reference partition to point to the deployed service object, and move the service object from the new partition into the reference partition. This tells iPlanet UDS to use the service object that the reference partition points to, rather than creating a new instance of the service object.

To create the reference partition, use the New Reference Partition command. When you choose the New Reference Partition command, a dialog displays a list of the applications from which you can select an existing service object. This list consists of all those applications for which application distributions have already been made. Therefore, you should make sure that the distribution that includes the service object you want to reference was made before you try to reference it.

➤ **To make a reference partition**

1. In the Logical Partition browser, select the service object name that you wish to access from the reference partition.

2. Choose the Component > New Reference Partition... command.

The Select Containing Application dialog opens.



3. In the Select Containing Application dialog, select the application that contains the existing service object you wish to reference, and click the OK button.

If automatic starting is specified for the environment (the default), the reference partition will automatically start the service object it references when this is necessary. If the automatic starting property is turned off for the environment (check with your system manager), you can turn on automatic startup for the service object in the reference partition by using the (a) option in its environment search path. However, before using this auto-start feature, please check with your system managers, because auto-starting an individual service object affects the startup behavior of the entire application.

For complete information about auto-starting partitions, see [“Specifying Auto-Start for a Partition” on page 434](#).

When the application that contains the reference partition is installed, the application that contains the service object that is being referenced must also be installed. If the application that contains the service object is not installed, the application that contains the reference partition will fail at runtime when it tries to access the service object.

If, in the future, the compatibility level of the application that defines the service object referenced by the reference partition is raised, and you want to use the new release of the service object, you must update your application appropriately. For information about using new compatibility levels for service objects, see [“Using Compatibility Levels to Upgrade” on page 478](#).

## Using Reference Partitions with Connected Environments

If your deployment environments are connected and a service object that you need to access can run only in one of the environments, you can access that service object from other environments by using a reference partition combined with an *environment search path*. (To see if a given environment is connected to any other environments, you must use the Environment Console or Escript. For information on how to do so, see the *iPlanet UDS System Management Guide*.)

In a reference partition to a connected environment, you use the environment search path on the service object to specify the name of the environment that contains the service object you want to access. When the environment search path for a service object specifies an environment name other than the current environment, every time the service object is referenced, iPlanet UDS always uses the service object in the specified environment.

When you have applications in multiple environments that need to use a specific service object in a specific environment, you should deploy the application that contains the service object in the environment in which it can run. Then, each application in the other environments can create a reference partition to the service object in that application. The search path from each application would include *only* the specific environment where that service object is deployed. (The search path must *exclude* the local environment).

Making a reference partition to a service object in a connected environment is similar to making a reference partition to a service object within the current environment, however, you must follow some extra steps. First, you must make a distribution for the application that contains the service object.

### ► To make the distribution for the application that contains the service object

1. Import the project that defines the service object into the development repository.

Remember, the project that defines the service object must have been exported from the repository where it was defined using the Fscript **ExportPlan** command with the **ids** option. If it was not, the application with the reference partition will not be able to inter-operate with the application that is running the service object. If your copy of the exported project does not include unique IDs, you need to get another copy of the project that does.

2. In the Project Workshop, choose the File > Configure as > Server command to partition the application that contains the service object.

3. In the Partition Workshop, partition the application appropriately and choose the File > Make Distribution command to make a distribution for the application.

Making the distribution makes the service object available for referencing in the current environment. However, you should *not* install the application, because the service object is intended to run outside of your environment and cannot be installed in your current environment.

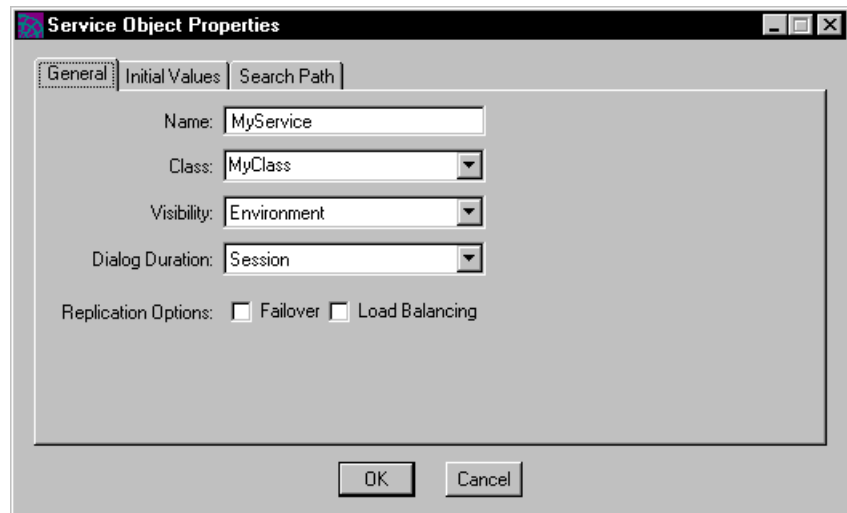
Second, you must use the service object's environment search path to specify which environment actually contains the service object to be accessed.

When a service object is in a reference partition and you provide an environment search path for the service object, iPlanet UDS searches the environments in the environment search path for a service object to use within the current application. iPlanet UDS uses the first service object it finds in the search path.

➤ **To create the reference partition with an environment search path**

1. Use the Component > New Reference Partition command to make a reference partition for the service object and assign it.
2. In the assigned reference partition, double-click the service object name.

The Service Object Properties dialog opens.



3. Click the Search Path tab.

4. On the Search Path page, enter the environment search path for the service object (see [“Specifying the Environment Search Path” on page 262](#) for the environment search path syntax).

Because you want to use a service object in a different environment, the environment search path should *not* include the current environment.

## Defining a Client Partition as an Applet

An applet is a client application that can be launched from another application using the AppletSupport library, but that cannot be run as an independent application.

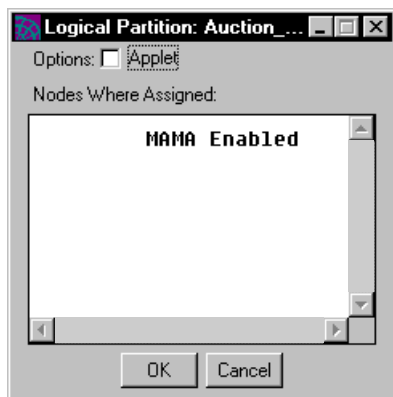
You can configure any client application as an applet by turning on the Applet toggle in the Logical Partition Properties dialog for the client partition.

See [“Writing Applications That Use the Launch Server and Applets” on page 435](#) for complete information on writing and configuring an application that launches applets.

### ► To create an applet

1. In the Logical Partition browser, double-click the client partition.

The Logical Partition dialog opens.



2. In the Logical Partition dialog, turn on the Applet toggle.



## Combining Service Objects and Partitions

When you partition your application for the first time, iPlanet UDS assigns the service objects to logical partitions according to the rules described under [“Creating a Default Application Configuration” on page 240](#). The Partition Workshop then allows you to move service objects between compatible partitions in many different ways.

The way you combine service objects onto partitions has a significant effect on the performance and behavior of your application. Some particularly useful combinations of service objects are:

- combining user-visible service objects onto one partition
 

This combination minimizes server process resources and network connection resources on clients and servers.
- combining environment-visible service objects onto one partition
 

This combination minimizes server process resources and network connection resources on clients and servers.
- combining an environment-visible service object and a user-visible service object
 

This combination make the user visible service object private to the partition containing the environment-visible service object.
- combining load-balanced service objects on a single partition
 

This combination allows you to minimize process resources and network connection resources in some cases.

The following sections provide detailed information about these basic combinations.

### *Combining User-Visible Service Objects On Partitions*

A partition that includes only user-visible service objects is a *private partition* for the client partition (or for any other server partition that is referencing one of the user-visible service objects). iPlanet UDS starts a new copy of the private partition each time a client or another server accesses the service object in the application.

By default, the Partition Workshop places all your user-visible service objects on your client partition, if possible. Otherwise, the user-visible service objects are on a private partition on whichever server node has the necessary resources.

By combining a number of user-visible service objects onto a single private partition, you minimize the number of server processes needed to support each client, as well as the network connection resources needed both on the client and the server. In addition, any access between the service objects within the partition is executed within the same process and is therefore very efficient.

### *Combining Environment-Visible Service Objects On Partitions*

A partition that includes only environment-visible service objects is a *shared partition*. All clients and servers that access the service objects in the partition share the partition. There is only one copy of a shared partition per application (unless it is replicated for load balancing or failover). The shared partition can be started up manually from the Environment Console or Escript, or it will be started automatically on first access.

By default, the Partition Workshop places all your environment-visible service objects in the same partition (to minimize the number of partitions) on a server node.

By combining a number of environment-visible service objects onto a single shared partition, you minimize the number of server processes needed to support the clients of the application, as well as the network connection resources needed both on the client and on the server. In addition, any access between the service objects within the partition is executed within the same process and is therefore very efficient.

However, if any of the environment-visible service objects are accessing a resource that supports only single-threaded access, such as a relational DBMS or a C project with the multi-threading option disabled, combining environment-visible service objects into a single partition can cause poor interactive performance in the clients. This is because access from *one* client to the single-threaded resource will delay processing for *all other* clients that are accessing *any* of the service objects in the shared partition. Therefore, to provide the best performance for a shared service object that accesses single-threaded resource, we recommend placing the shared service object in its own partition, possibly a partition that is replicated for load balancing.

### *Combining Environment-Visible with User-Visible Service Objects*

A partition that includes both an environment-visible service object and a user-visible service object makes the environment-visible service object shared for the application and the user-visible service object *private for that partition*. The user-visible service object can be accessed indirectly *through* the environment visible service object—neither the client partition nor any other service objects can access it directly.

The Partition Workshop does not include environment-visible and user-visible service objects within the same partition by default. However, you can move a user-visible service object onto the same partition with an environment-visible service object.

This combination is very useful when you have an environment-visible TOOL class service object that provides DBMS access through a user-visible DBSession or DBResourceMgr service object. We recommend placing the TOOL class service object on the same partition with the DBSession or DBResourceMgr service object because this way a client can access the database only through the TOOL service object. The DBSession or DBResourceMgr service object is protected from inappropriate access, because no other partitions can access it.

### *Combining Load-Balanced Service Objects*

When you combine more than one load-balanced service object on a single partition, iPlanet UDS automatically places all the routers for the load-balanced service objects onto a single router partition.

This architecture is useful for minimizing the number of server processes needed to support an application, because only one process is needed to support both routers for the application. You can use this architecture if you have several service objects that need to access multi-threaded resources, such as:

- Service objects that access true multi-threaded C projects. The C projects must have the multi-threaded option enabled, and must be implemented so they support the multi-threaded conventions on the platform.
- TOOL class service objects that do not access single-threaded resources (such as a relational DBMS).

However, if any of the load-balanced service objects access a resource that supports only single-threaded access, such as a relational DBMS or a C project that has not enabled the multi-threaded option, combining load-balanced service objects onto a single partition can lead to uneven and unpredictable interactive performance in the clients. This is because the two routers perform their routing completely independently from one another. A client request on *one* of the routed service objects can delay a concurrent client request on the *other* routed service object, even though it is from an unrelated client. Because of this behavior, we do not recommend this architecture for service objects that access relational DBMSs or single-threaded external services.

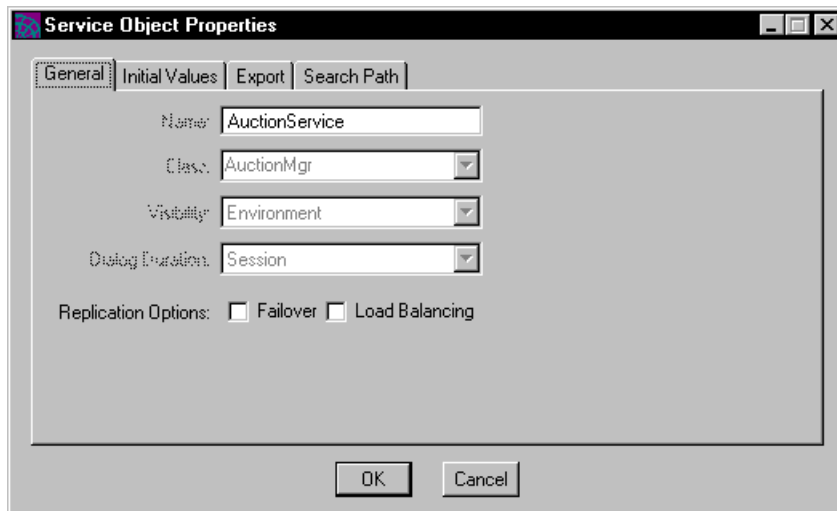
## Modifying a Service Object Definition

In the logical partition, you can modify certain properties of the service object. Note, however, that changing the service object definition in the Partition Workshop applies only to the current configuration. Changes do not affect the original service object definition that you specified in the Project Workshop.

In addition, if the service object is in a reference partition, you cannot modify its definition in the current configuration. You can only make changes to the service object definition from the configuration that contains the original service object definition. The exception to this is the environment search path. You must specify a new environment search path for the service object in a reference partition; the environment search path in the original service object definition is ignored in the current configuration.

➤ **To modify a service object definition**

1. Double-click the service object name to open the Service Object Properties dialog, as shown below.



## 2. Update the appropriate properties.

In the Service Object Properties dialog, you can update only certain properties. The following table describes the properties on each tab page that you can update for each kind of service object.

Service object	Properties you can update
DB Resource Manager	General Tab page: Failover (see below), Load Balancing (see below) Database Tab page: Database Manager Search Path Tab page: Search Path (see below)
DB Session	General Tab page: Failover (see below), Load Balancing (see below) Database Tab page: Database Manager, Database Name, User Name, Password Search Path Tab page: Search Path (see below).
User	General Tab page: Failover (see below), Load Balancing (see below) Export Tab page: Export Name (see below), External Type (see below) Initial Values Tab page: initial values for attributes Search Path Tab page: Search Path (see below)

Use the Service Object properties dialog to modify the definition of an unassigned service object. After you change the external manager name to a name that is available in the current environment, iPlanet UDS will automatically reassign the service object to the appropriate partition. When all unassigned service objects are assigned appropriately, you can run the application.

## Failover and Load Balancing

You can use the Failover and Load Balancing options to turn on replication for a service object. Note, however, that when you turn on failover and load balancing for a service object, iPlanet UDS does not automatically replicate your service objects for load balancing or failover. Instead, you must replicate the partitions to which the service objects are assigned. You can replicate a partition either by assigning it to an additional node (as described under [“Adding Partition Assignments” on page 264](#)) or by setting the startup replicates for an individual assigned partition on a single node (described under [“Changing Assigned Partition Properties” on page 265](#)).

## Setting the Export Name and External Type

The Export Name and External Type properties are for use with the following external systems:

- DCE
- Encina
- IIOP
- ObjectBroker
- OLE
- XML services

If your service object is not going to function as a server for one of these external systems, you should ignore these properties. See *Integrating with External Systems* for information about integrating DCE, Object Broker, OLE, Encina, and XML services with iPlanet UDS applications. See *iPlanet UDS Java Interoperability Guide* for information about integrating with IIOP.

The export name is the name to be used by the client to identify the service object.

The external type specifies the type of external client application that can access this service object, either DCE, ObjectBroker, OLE, IIOP, Encina, or none. This external type tells iPlanet UDS to export the files that you need to set this service object as a server for the specified external system.

For example, specifying DCE as the external type tells iPlanet UDS to export an interface definition file when you make a distribution. You will later use the file to set up this service object as a DCE server. The programmer writing DCE client applications that access this service object will also read this file to understand how to access the methods in this service object.

## Specifying the Environment Search Path

The environment search path for the service object specifies the connected environments in which iPlanet UDS searches for the service object (see the *iPlanet UDS System Management Guide* for complete information on connected environments).

The environment search path you specify for an individual service object within a configuration overrides the environment search path in the original service object definition itself specified in the Project Workshop (it does not add to it). You can use this feature only if the current environment is connected to other iPlanet UDS environments (see the *iPlanet UDS System Management Guide* for information).

If you use the environment search path for a service object, you should be sure to document exactly what you specified for your system manager. Setting the environment search path for an individual service object overrides what the system manager specifies for the environment as a whole, and the system manager needs to be aware of your changes.

To specify the environment search path, enter a string that includes one or more environment paths. iPlanet UDS searches for the service object in the same order as the paths in the string. A blank environment search path specifies using the default search path.

### **Syntax**

The syntax of the search path string is:

*path* [(a)] [: *path* [(a)]...

*path* is:

(@ | @*environment\_name*)

A special (a) option allows you to specify that the service object identified by a specific path should automatically be started if necessary. However, before using this auto-start feature, please check with your system managers, because auto-starting an individual service object affects the startup behavior of the entire application. For complete information about auto-starting service objects, see [“Specifying Auto-Start for a Partition” on page 434](#).

You can use an environment variable to specify the contents of the environment search path. In fact, we recommend using an environment variable because this makes it possible to change the environment search path for the service object after the application is deployed.

The value for the environment variable is set on first access to the service object, using the value of the environment variable as set on the service object’s partition.

### **Syntax**

$\${environment\_variable\_name}$

Be sure to include the braces!

The following example illustrates a search path that looks first in the current environment, second in the “la” environment, and last in the “sf” environment:

```
@:@la:@sf
```

If two or more connected environments share the same environment name, and these environments are specified in the environment search path, then you must use the environment UUID to specify each environment. Specify this environment UUID in place of the `@environment_name`, for example, `B763E430-22FF-11D0-A5AA-5BC569EDAA77`. For more information about the environment UUID, see the *iPlanet UDS System Management Guide*.

## Changing Partition Assignments

You can modify a partition assignment in the configuration by:

- adding a partition assignment
- moving an assigned partition from one node to another
- deleting an assigned partition from the node

### Adding Partition Assignments

To assign a logical partition to a node, you simply drag the logical partition onto the appropriate node. The node must provide the resources necessary to run the particular partition.

If the logical partition is replicated, the new assigned partition will be enabled. If the logical partition is non-replicated and there is already a copy of the logical partition assigned in the environment, the new assigned partition will be disabled.

#### ➤ To assign a logical partition

1. In the Logical Partitions browser, select the logical partition you wish to assign.
2. Drag the logical partition to the node to which you wish to assign it.

### Moving Partitions

You can move an assigned partition to any compatible node. To be compatible, the node must provide the resources required by the service objects on the assigned partition.

To move a partition from one node to another, simply drag the partition to its new location. If the node is incompatible, iPlanet UDS will display an error dialog explaining why the service object cannot be moved.



## Deleting Partitions

You can delete any disabled, assigned partition from any node. (The last enabled copy cannot be deleted.)

- **To delete a disabled, assigned partition**
  1. Select the assigned partition.
  2. Choose the Edit > Delete command.
  3. Confirm that you wish to delete the partition.

## Changing Assigned Partition Properties

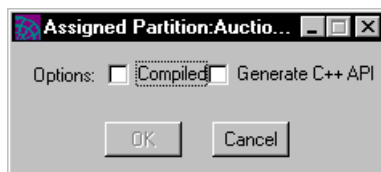
To set the properties for an assigned partition, double-click the assigned partition name, or select the assigned partition and choose the Component > Properties... command. This command opens the Assigned Partition dialog, where you can set the properties that are appropriate for the particular partition.

The following two sections describe the different properties available for the assigned client and server partitions.

### Assigned Client Partition Properties

Figure 8-6 shows the properties available for an assigned client partition:

**Figure 8-6** Assigned Partition Properties Dialog (Client Partition)



By default, a client partition is a standard partition. If you wish to make the partition a compiled partition, turn on the Compiled toggle. Remember, after specifying that the partition is compiled, you may need to perform extra steps to produce the application distribution. See [“The Make Distribution Command and Compiled Partitions” on page 279](#) for information.

Note that even though the Assigned Partition dialog shows the Compiled option for all client platforms, code generation for client partitions is not supported for all platforms. The following table summarizes:

Platform	Client Code Generation Available?	Autocompile Support?
Windows NT	Yes	Yes
UNIX	Yes	Yes
OpenVMS	Yes	Yes

For an assigned client partition, you can request that iPlanet UDS generate and compile files for a C++ API. To generate the C++ API for a particular assigned client partition, turn on both the Compiled and Generate C++ API properties on the Assigned Partition Properties dialog.

When you later use the automatic compilation feature of the Make Distribution command or the fcompile command to compile the partitions, iPlanet UDS generates and compiles the files for the C++ API. See *Integrating with External Systems* for step-by-step instructions for generating a C++ API for an iPlanet UDS application.

## Assigned Server Partition Properties

Figure 8-7 shows the properties available for an assigned server partition:

**Figure 8-7** Assigned Partition Properties Dialog (Server Partition)



The exact properties available on this dialog depend on whether or not the partition is replicated.

By default, a server partition is a standard partition. If you wish to make the partition a compiled partition, turn on the Compiled toggle. Remember, after specifying that the partition is compiled, you may need to perform extra steps to produce the application distribution. See [“The Make Distribution Command and Compiled Partitions” on page 279](#) for information.

The Disabled toggle indicates whether the assigned partition is currently disabled. To enable a disabled partition, you can turn off this toggle. To disable an enabled replicated partition, turn on this toggle.

Only one copy of a non-replicated partition can be enabled. Therefore, if you enable a non-replicated partition, the other assigned copies of that partition will automatically be disabled. To disable a non-replicated partition, you must explicitly *enable* another copy of that partition.

The Thread Package property specifies the thread package used by the partition. The default thread package depends on the particular platform (described in the *iPlanet UDS System Installation Guide*). If you do not want to use the default thread package, use the Thread Package property to specify one of the other thread packages supported for the particular platform. See the *iPlanet UDS System Installation Guide* for information about which thread packages are supported for each platform.

To specify startup flags for the server partition, enter a list of flags in the Server Arguments field. These can be the following:

Flag	Description
<b>-fm</b> <i>memory_flags</i>	Specifies the space to use for the memory manager. See <i>A Guide to the iPlanet UDS Workshops</i> for information.
<b>-fl</b> <i>logger_flags</i>	Specifies the logger flags to use for the session. This has no effect for standard partitions. See <i>A Guide to the iPlanet UDS Workshops</i> for information on logger flags.

You can also include your own application-specific flags. See the `CmdLineArgs` attribute of the `Partition` class in the Framework Library online Help for information.

The syntax for the server arguments is platform dependent. For example, any argument with parentheses needs to be enclosed in double quotes or there will be a runtime error.

---

**NOTE** The server arguments take effect only when the partition is started using the Partition Workshop, the Environment Console, or Escript; these arguments are ignored when the partition is started manually using the **ftexec** command.

---

To provide startup replicates for a replicated partition, enter the total number of partitions to be started in the Replication Count field. When the application starts, iPlanet UDS automatically creates the number of replicates you specify (unless the system manager overrides your setting).

## Changing Configuration Properties

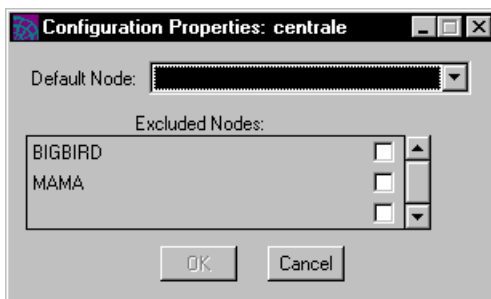
For the configuration in general, you can set the following two properties:

Property	Description
Default node	iPlanet UDS automatically assigns all partitions that can run on it to the default node.
Excluded nodes	iPlanet UDS will not use these nodes in the configuration.

Changing the node settings causes the application to be automatically repartitioned after you click the OK button on the properties dialog.

### Viewing and Setting the Configuration Properties

To view the default node and excluded nodes for the current configuration, choose the File > Properties... command. The Properties... command opens the Configuration Properties dialog, shown in [Figure 8-8](#), displaying all the nodes in the environment. If the toggle next to the node name is checked, this indicates that the node is excluded from the configuration.

**Figure 8-8** Configuration Properties Dialog

To specify a default node, select the node name from the drop list. To exclude a node, turn on the toggle next to the node name.

## Recreating the Default Configuration

The File > Repartition... command erases all changes you have made to the configuration and forces iPlanet UDS to perform a complete automatic partitioning on the application. A complete partitioning removes all current assignments, and then recreates the default partitioning scheme from scratch.

You typically do not need to use the Repartition... command. Repartitioning is useful mainly when you have made changes to the configuration that you wish to remove.

When you choose the Repartition... command, iPlanet UDS displays a confirmation dialog to ensure that you do not accidentally erase your changes. When you confirm that you want to repartition the application, iPlanet UDS performs the repartitioning and displays the default configuration in your workshop window.

# Making an Application Distribution

This section describes how to create an iPlanet UDS application distribution for installation into an active iPlanet UDS environment. You create an application distribution from a configuration using the File > Make Distribution command in the Partition Workshop.

The application configuration does not include the libraries that are needed by the application. If your application uses one or more libraries, you must be sure that they are installed in all intended deployment environments. First, you must create a separate library configuration for the libraries, as described under **“Modifying a**

[Library Configuration](#)” on page 308, and then create a separate library distribution for them as described under [“Making a Library Distribution”](#) on page 312. Both the application distribution and library distribution must be installed for the application to run. See [“Installing an Application Distribution”](#) on page 286 and [“Installing a Library Distribution”](#) on page 317 for information about installing applications and libraries.

## Understanding Application Distributions

As the last step in the partitioning process, you create an *application distribution*. An application distribution is a collection of files outside of the development repository that represent an application intended for deployment. Once you create an application distribution, you load the distribution into a target environment, and install it using iPlanet UDS system management tools. Additional steps are required if your configuration contains compiled partitions and you do not or cannot use automatic compilation when you give the Make Distribution command.

### Standard Partitions

A standard partition, with its companion runtime repository, runs against the **ftexec** executor (or one of its variants), which is part of every iPlanet UDS development or runtime system. Any partition, whether client or server, can be a standard partition. See the *iPlanet UDS System Management Guide* for information about **ftexec** and its variants.

If your application uses only standard application partitions, your application distribution is complete when you create it (that is, after you give the Make Distribution command). You can install the application immediately into an active iPlanet UDS environment.

### Compiled Partitions

A compiled partition runs as its own independent executable in the context of an iPlanet UDS application. You designate a partition as compiled by opening the assigned partition’s properties dialog (see [“Changing Assigned Partition Properties”](#) on page 265).

Compiled client partitions are not supported on all the client platforms. The following table summarizes which platforms support client code generation and indicates whether automatic compilation (described below) is available:

Platform	Client Code Generation Available?	Autocompile Support?
Windows NT	Yes	Yes
UNIX	Yes	Yes
OpenVMS	Yes	Yes

If your configuration contains compiled partitions, you may wish to select the automatic compilation option when you give the Make Distribution command. Automatic compilation will be available if your system manager has set up your iPlanet UDS installation with the appropriate applications. See the *iPlanet UDS System Management Guide* for information on this.

If automatic compilation is not available, or if for some reason you choose not to use it, you must use the iPlanet UDS `fcompile` command to generate C++ code for the partitions and compile the code after you make a distribution. This process is described under [“The Make Distribution Command and Compiled Partitions” on page 279](#).

## Launching Applets and Other Applications

If your client application launches one or more applets, you must create a separate application distribution for each applet that the main application launches. Likewise, if your client application launches other client applications, you must create a separate distribution for each application that is launched (if the distributions have not already been created).

To ensure that all the applications and applets required by a main client application are installed when the main application is deployed; do the following:

- include all the application distributions with the main client applications
- write an Escript script that installs all the applications and applets when the main client application is installed

See [“Writing Applications That Use the Launch Server and Applets” on page 435](#) and the *iPlanet UDS System Management Guide* for complete information on writing and configuring applications that use the Launch Server and applets.

## Adding an Icon File for Windows to the Distribution

You can include in your application distribution icon files that iPlanet UDS will use when creating the client icon on Windows NT. After using the Make Distribution command to generate the application distribution, add the appropriate .ico files to the port-specific directories in the application distribution directory structure shown in the following section.

## Application Distribution Directory

An application distribution consists of several elements, including the actual partitions you assigned, and a group of files describing the application.

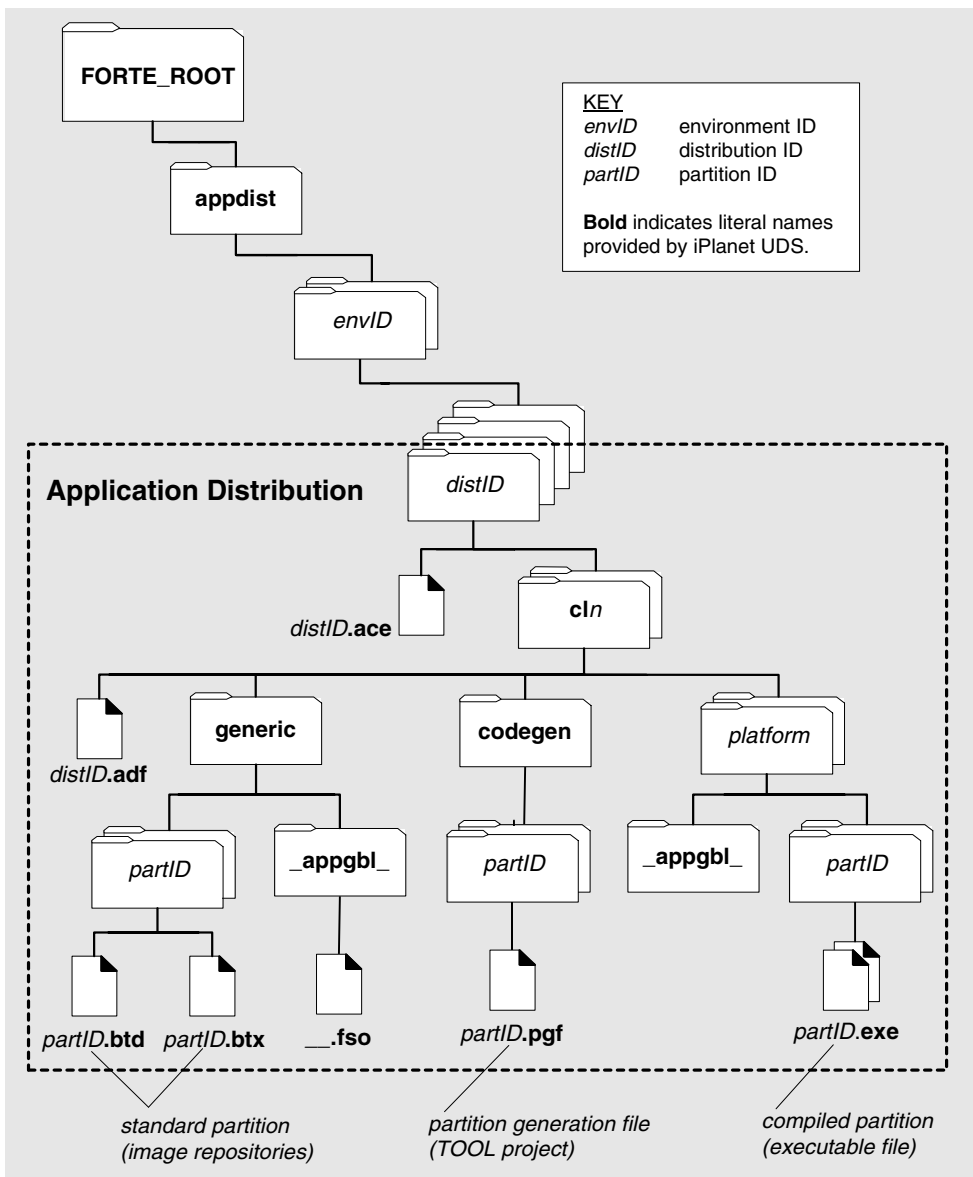
The distribution is contained in the following directory:

**FORTE\_ROOT/appdist/*environment\_ID***

The *environment\_ID* is the ID for the deployment environment, the *distribution\_ID* is the first eight characters of the name of the project that was configured. The *n* in the *cln* is the compatibility level of the project that was configured.



**Figure 8-9** Application Distribution Directory Structure



The following table describes these files.

Name	Purpose
—.ace	File which maps iPlanet UDS application component names to unique identifier names
—.adf	Application distribution file. Contains information about the application (partitioning) configuration—what partitions are assigned to what nodes—for use in the deployment process. (The .adf extension is added to the application's unique identifier name.)
generic	Directory that contains all portable files. The partition directories below the generic directory contain all the standard partitions.
codegen	Directory that contains all source files used for generating C++ compiled partitions.
platform1...	Directories that contain non-portable compiled partitions for each supported platform.
_appgbl_	Directory in which you or iPlanet UDS can place files that would be installed along with any partition (generic directory) or set of partitions assigned to a node (platform directories). For example, this file is used for message files that are used for internationalization.
—.btd, —.btx	Standard partition image repositories. (The .btd and .btx extensions are added to the partition's unique identifier name.)
—.pgf	Partition generation file. Contains the source code used to make compiled partitions for a single logical partition. (The .pgf extension is added to the partition's unique identifier name.)
—.exe, ...	Compiled partition executable file.(The name is platform dependent and based on partition's unique identifier name.)

## File Naming Conventions

In iPlanet UDS distributions, all names are based on the name of the main project for the application. The following table describes how the names of various distribution components are determined:

Name	Determined by
Application name	The name of the project that was configured.
Full project and application names	The name of the project that was configured with the compatibility level appended to the end. For example, if a main project named "Auction" has a compatibility level of 1, its full name is "Auction_cl1". The full application name is the same.
Distribution ID	<p>The first 8 characters of the name of the project that was configured, not the entire project name (because name length is limited on some platforms).</p> <p>Standard partitions, runtime repositories, compiled partitions, distribution information files, application distribution files, and even some of the directory structure that defines the application distribution all use the distribution ID as a basis for their names.</p>
Partition names	The full application name plus an iPlanet UDS system-generated number. For example, the name of a partition for the "Auction_cl1" application is "Auction_cl1_part#" (where # is the system-generated number).
Partition unique identifier	The first 6 characters of the application name plus the partition number. The partition generation file (.pgf) is derived from the partition's unique identifier.

For information about project names, see *A Guide to the iPlanet UDS Workshops*. For information about environment names, see the *iPlanet UDS System Management Guide*.

## Using the Make Distribution Command

The Make Distribution command produces an application distribution from the current configuration.

The following table briefly describes the options available for the Make Distribution command:

Make Distribution Option	Description
Local/Remote	Specifies whether the distribution should be made on the local node or a remote node. If you choose remote, you can select the specific node where you wish to create the distribution. See <a href="#">“Local/Remote Option” on page 277</a> for further information.
Auto-Compile	Specifies whether partitions marked as compiled should be automatically compiled while making the distribution. If auto-compile is off, you must compile the partitions yourself as described under <a href="#">“The Make Distribution Command and Compiled Partitions” on page 279</a> .  Automatic compilation will work only if your system manager has set up your iPlanet UDS installation correctly, and is not available on all platforms for client partitions.
Full or Partial Make	If a distribution already exists for the configuration, a partial make creates the distribution only for those components that have changed since the last make. See <a href="#">“Full or Partial Make Options” on page 277</a> for further information.
Install in Current Environment	If the configuration is for the current environment, you can automatically install. However, you should avoid installing too many versions of the same application. See <a href="#">“Install in Current Environment Option” on page 278</a> for further information.
Include Source	When you are making a library distribution, this option specifies that the library source code will be included in the export files.

When the Make Distribution command completes, a message in the status line indicates that the distribution was complete. You will be notified if automatic compilation or installation fails.

The following sections provide more detailed information about the Make Distribution command’s options.

## Local/Remote Option

When you make an application distribution from the Partition Workshop, the first option on the Make Distribution dialog is where to store the application distribution: on the local machine or on a remote server.

If you are running the Partition Workshop on a model node, do not select Local as the node where the application distribution is placed. The Environment Manager never searches for application distributions on model nodes, therefore, you should not make an application distribution on a model node. Instead, select the remote node where you chose to store all application distributions. Otherwise, the Environment Manager cannot locate the application distribution.

## Auto-Compile Option

In general, if your system manager has set up your environment to support automatic compilation, you can use the Auto-Compile option to compile your partitions automatically.

## Full or Partial Make Options

The first time you make a distribution for a given configuration in a given workspace, iPlanet UDS creates a full distribution, whether or not you select the Full or Partial Make options. The next time you make a distribution for the same configuration in the same workspace and you have not yet given an Update Workspace command, you can request a partial make. For a partial make, the Make Distribution command changes the existing distribution by updating only those components that have changed. Naturally, you should only choose the Partial Make option after you have already created a full distribution.

To use a partial make, you must not only make the distribution from the same workspace that you used to make the original distribution but you must also use the same *node*. When you request a partial make, iPlanet UDS checks the node to make sure that the appropriate full distribution is already present, and if it is not, you will get an error.

If you have already made a distribution that contains a compiled partition, and you want to make a distribution again for the same application, you should consider the following situations when you decide whether to make a partial distribution or a full distribution.

- You have changed a partition from a standard, or interpreted, partition to a compiled partition.

You can use the Partial Make option with the Auto-Compile option, and the new distribution will contain the newly-compiled partition.

- You tried to make a distribution with the Auto-Compile option, and the compile failed.

After you have corrected the cause of the compilation error, you need to select the Full Make option with the Auto-Compile option to auto-compile the compiled partitions of the distribution.

- You made a distribution that contained a compiled partition without using the Auto-Compile option, and you now want to make a distribution using the Auto-Compile option.

You need to select the Full Make option with the Auto-Compile option to auto-compile the compiled partitions of the distribution.

## Install in Current Environment Option

The Install in Current Environment Option lets you automatically install the application in the development environment after the distribution is created.

By default, iPlanet UDS assumes that after you have installed an application, the configuration of the installed application is the correct configuration for the application.

This assumption can affect whether or not your changes to the application configuration in the Partition Workshop are used when installing the application.

If you change the contents of any logical partition, iPlanet UDS cannot re-install the application because the new configuration is incompatible with the configuration of the installed application.

If you have changed only the partition assignments of an application that is installed in the current environment, the installed configuration will remain the same as for the original installation of the application, no matter how you assign the logical partitions in the environment after you have installed the application. Therefore, to see the changes, you must first shutdown and un-install the installed application.

### ➤ **To change the configuration of an installed application without changing the contents of any logical partitions**

1. *Shutdown* and *un-install* the application using the Environment Console or Escript.

You might need to ask your system administrator to perform this task, depending on who is permitted to use the Environment Console or Escript in your environment. For information about how to un-install an application, see [“Removing an Application or Library” on page 319](#).

2. In the Partition Workshop, configure the application the way you want, then choose the File > Make Distribution command.

In the Make Distribution dialog, choose the Full Make and Install in Current Environment options. You can also choose the Auto-Compile option, if appropriate.

3. Click the Make button.

## The Make Distribution Command and Compiled Partitions

If your configuration contains one or more compiled partitions and you did not use automatic compilation with the Make Distribution command, the process for making a distribution has some extra steps.

### ► To make a distribution for a configuration containing compiled partitions

1. Give the appropriate Make Distribution command.

For each compiled partition, iPlanet UDS creates a partition generation file (.pgf file), which you use with the `fcompile` command to generate a completed, compiled partition.

2. Set your environment variables and path as described under “[Environment Variables and Path](#)” on page 279.
3. Run the `fcompile` command, as described under “[Using the fcompile Command for Compiled Partitions](#)” on page 281.

### Environment Variables and Path

Depending on certain characteristics of your system, you may need to set a few environment variables before you make the distribution for a compiled partition. You also have to make sure you have a valid C++ compiler installed in your path (see the *iPlanet UDS System Installation Guide* for information about which version of the C++ compiler you need).

The following table describes the environment variables that affect the fcompile command:

Environment Variable	Description
ORACLE_HOME	Set the ORACLE_HOME environment variable if you are running Oracle in the partition. Set the variable to the root directory of the Oracle installation. If you do not set this environment variable, you will receive an error message.
SYBASE	Set the SYBASE environment variable if you are running Sybase in the partition. Set the variable to the root directory of the Sybase installation. If you do not set this variable when necessary, you will receive an error message.
FORTE_X_LIBDIRS	Set this variable only if the Display library is contained in the partition and if you are unsure if your X Window header files reside in a standard location. Generally, you should set this variable if you are running on a Sparc; if your hardware vendor also supplied your windowing software, then you probably do not need to set the variable. The syntax is:  FORTE_X_LIBDIRS -L/ <i>dir_name</i> /lib
FORTE_X_HEADERDIRS	Set this variable if the Display library is contained in the partition. This variable points the C++ compiler to the directory in which the header files for X Windows and Motif are stored. The syntax is:  FORTE_X_HEADERDIRS -I/ <i>dir_name</i> /include  To find out if the Display library is in the server partition, use the <b>FindProj</b> , <b>FindApp</b> , and <b>ShowApp</b> commands in Fscript.
FORTE_CG_RESERVED	Set this variable to provide a file containing reserved words to supplement the list of iPlanet UDS reserved words. You should provide this file when your project uses names for class components that are already reserved by your C++ compiler. The iPlanet UDS code generator uses the reserved words file to rename the class components in order to avoid conflicts.  The reserved word file is strictly optional—you do not need to provide one. If you do not set the variable, the default is: \$FORTE_ROOT/install/scripts/cgreserv.lst. The syntax is:  FORTE_CG_RESERVED <i>file_specification</i>



## Using the `fcompile` Command for Compiled Partitions

The `fcompile` command generates code, compiles, and links a compiled partition from a `.pgf` file. The process for compiling a client partition and a server partition is the same.

### Portable syntax

```
fcompile [-c component_generation_file] [-d target_directory] [-o output_file]
          [-cflags compiler_flags] [-lflags linking_flags]
          [-fns name_server_address]
          [-fm memory_flags] [-fst integer] [-fl logger_flags] [-cleanup]
```

### OpenVMS syntax

#### VFORTE FCOMPILE

```
[/COMPONENT=component_generation_file]
[/DIRECTORY=target_directory]
[/OUTPUT=output_file]
[/COMPILER=compiler_flags]
[/LINKING=linking_flags]
[/NAMESERVER=name_server_address]
[/REPOSITORY=repository_name]
[/MEMORY=memory_flags]
[/LOGGER=logger_flags]
[/CLEANUP]
```

The `fcompile` command has other flags that are for use only when integrating with external systems. See *Integrating with External Systems* for information on these flags.

The following table describes the command line flags for the `fcompile` command:

Flag	Description
<code>-c</code> <i>component_generation_file</i> <code>/COMPONENT=</code> <i>component_generation_file</i>	Specifies the file that iPlanet UDS compiles. This value includes the path where the file resides if the file is not in the current directory. By default, iPlanet UDS compiles all files in the current directory.

Flag	Description
<b>-d</b> <i>target_directory</i> <b>/DIRECTORY=</b> <i>target_directory</i>	Specifies where the compiled directories will be placed. By default, <b>fcompile</b> compiles files in the current directory, and places the compiled files in the current directory.  <i>target_directory</i> is a directory specification in local syntax.  If the <b>-c</b> ( <b>/COMPONENT</b> ) flag is also specified, the <b>-d</b> flag specifies where the compiled component files will be placed. Otherwise, the directory specified by the <b>-d</b> ( <b>/DIRECTORY</b> ) flag specifies both the directory containing the files to be compiled and the directory where the compiled files will be placed.
<b>-o</b> <i>output_file</i> <b>/OUTPUT=</b> <i>output_file</i>	The log output file used by the <b>fcompile</b> command.
<b>-cflags</b> <i>compiler_flags</i> <b>/COMPILER=</b> <i>compiler_flags</i>	Specifies any C++ compiler options. Any header file specifications included here are used before the specifications included in the C project definition. For more information about these options, see <i>Integrating with External Systems</i> .
<b>-lflags</b> <i>linking_flags</i> <b>/LINKING=</b> <i>linking_flags</i>	Specifies any linking flags. Any files included here are linked before files specified in the extended properties of the C project definition. For more information about specifying linking flags in the C project, see <i>Integrating with External Systems</i> .
<b>-fm</b> <i>memory_flags</i> <b>/MEMORY=</b> <i>memory_flags</i>	Specifies the space to use for the memory manager. See <i>A Guide to the iPlanet UDS Workshops</i> for information.
<b>-fst</b> <i>integer</i> <b>/STACK=</b> <i>integer</i>	The thread stack size in bytes for iPlanet UDS and POSIX threads. This specification overrides default stack size allocation. For more information, refer to the <i>iPlanet UDS System Management Guide</i> .
<b>-fl</b> <i>logger_flags</i> <b>/LOGGER=</b> <i>logger_flags</i>	Specifies the logger flags to use for the command. See <i>A Guide to the iPlanet UDS Workshops</i> for information.
<b>-cleanup</b> <b>/CLEANUP</b>	Deletes all the files except for the newly compiled partitions.

When you give the **fcompile** command for a given partition, you must do so on machine with the same architecture as the node to which the partition is assigned. For example, if you are building a compiled partition for an RS6000, you must run **fcompile** on an RS6000.

You cannot use a model node to create a distribution. Therefore, you should not run the **fcompile** command on a machine that is defined as a model node within the current environment.

► **To run fcompile for a compiled partition**

1. If necessary, move the .pgf file to the machine with the same architecture as the intended partition.
2. Set the necessary environment variables, as described in [“Environment Variables and Path” on page 279](#).
3. Give the **fcompile** command as described above.

```
fcompile -c aucio1.pgf -d ./rs6000
```

This command creates an executable partition file in the format used by the specific platform.

4. Return the executable partition file to the component’s subdirectory of the appropriate port in the distribution directory.

For OpenVMS only, you must ensure that the .pgf file’s record format is stream\_lf (you can check the format with the dir/full command). If the record format is not stream\_lf, or if you get an error reading the file, use the following command to change the file format:

```
set file/attrib=rfm=stmlf 'filename'
```

Your application distribution is now ready to install. For instructions on installing an application distribution, see [“Installing an Application Distribution” on page 286](#).

## Compiling a Partition for Use on Several Computing Platforms

You may need to compile a server for use on more than one computing platform. For example, let’s say you are compiling a server partition for an application that is load balanced across server platforms of varying architectures: one server is an IBM RS/6000, one is a Sun SPARCStation running UNIX, and a third is a VAXStation running VMS. In this case, you would compile three separate executable partition files using the same .pgf file.

## Packaging an Application Distribution

Once an application distribution is complete, it is ready to be packaged, along with any library distributions that must accompany it, for deployment.

A distribution for a compatibility level “#” is packaged for deployment by copying the `distribution_ID` directory—and any files it contains—plus the `cl#` directory, and everything contained below it, to a distribution medium such as a magnetic tape.

You can copy this branch of the directory structure using the **tar** command on UNIX platforms and the **backup** command on OpenVMS.

Any library distributions needed for the application to execute must also be packaged and included with the application distribution.

### Installing Additional Files with Your Application Distribution

When you prepare to install an application or library, you can include additional files, such as help files or release notes. You need to put the additional files in specific subdirectories of the `FORTE_ROOT/appdist` directory. The exact location of these files depends on your answers to the following questions:

- On what platforms will these additional files be installed?
- What components of the distribution (partitions or libraries) do these additional files belong with?

After you have added the files to the appropriate location in the application distribution directory, iPlanet UDS’s installation program can automatically install these files in the appropriate location along with the application or library distribution.

The following table explains where to place files that you want to add to the application or library distribution for installation, depending on where you want the additional files installed. The directory locations are all under the following directory:

`FORTE_ROOT/appdist/environment_ID/distribution_ID/cl#`

	<b>All Components</b>	<b>Specific Component</b>
All Platforms	<code>/generic/_appgbl_</code>	<code>/generic/component_ID</code>
Specific Platform	<code>/platform_ID/_appgbl_</code>	<code>/platform_ID/component_ID</code>

*platform\_ID* represents the iPlanet UDS identifier for a platform, for example, IBM\_AIX represents AIX.

*component\_ID* represents one of the following:

- a partition ID for a partition if the distribution is an application distribution
- a library ID if the distribution is a library distribution

For more information about the directory structures of application and library distributions, see [“Application Distribution Directory” on page 272](#) and [“About Library Distributions” on page 312](#).

For example, suppose you have an application in the MyEnv environment whose distribution ID is myapplic. If you have release notes that you want to have automatically installed with all partitions on all platforms, you should place the file in the following directory:

FORTE\_ROOT/appdist/MyEnv/myapplic/cl0/generic/\_appgbl\_

## Documenting a Distribution

Every application distribution should be accompanied by sufficient documentation to enable you to deploy the application in your environment and to troubleshoot runtime problems.

The following list covers the most important information that should be documented by developers for an application distribution:

- the library distributions required for the application to execute (if any)
- the function each partition performs
- the node architecture(s) each compiled partition requires
- the external resources each partition requires, by exact name
- the restricted libraries each partition requires (if any)
- the external object system (DCE, ObjectBroker, OLE) each partition requires (if any)
- the object memory allocation each partition requires
- the logger flags that should be used in troubleshooting the application
- the names and purpose of any files the developer chose to place in any \_appgbl\_ directory

- any environment variables that must be defined for partitions of the application
- environment search paths that have been set for any service objects in the application
- whether or not the application or library is compatible with any previous releases of the application or library

## Installing an Application Distribution

Installing an iPlanet UDS application in your target deployment environment consists of four basic steps, which are described in this section.

➤ **To install an application distribution**

1. Transfer the application distribution to a server node in your deployment environment.
2. Load the application distribution into your deployment environment's environment repository.

You can use the Environment Console or Escript utility to do this.

3. Modify the application's partitioning configuration, if you wish.

You can assign partitions to additional nodes and modify a number of assigned partition properties.

4. Install the application in your deployment environment.

iPlanet UDS automatically downloads each partition to the node (or nodes) in your environment on which it is supposed to run.

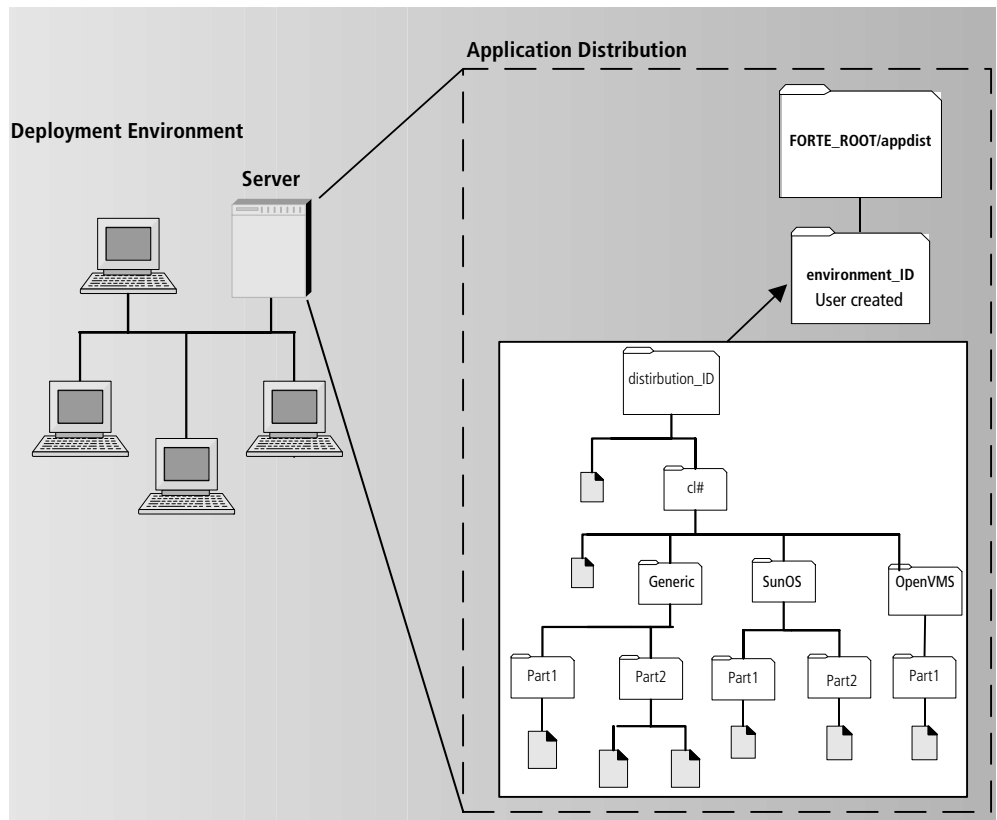
Before you try to run your newly installed application, make sure that any library distributions required by the application have also been deployed (see [“Installing a Library Distribution” on page 317](#)) and that any reference partitions required by the application have been started.

The deployment steps are discussed in the following sections.

## Transferring a Distribution to a Deployment Environment

Transferring a distribution to an iPlanet UDS deployment environment consists of copying the contents of the *distribution\_ID* directory, and the required *cl#* directory structure beneath it, to any iPlanet UDS server node you wish in your deployment environment. Do not copy the distribution to a client node and try to load the distribution; the Environment Manager cannot locate a distribution on a client node. You can use whatever copying technique you prefer (such as network copying utilities or copying onto tape or diskette).

**Figure 8-10** Transferring a Distribution to a Deployment Environment



Copy the distribution, as shown in [Figure 8-10](#), to the following iPlanet UDS directory structure on the selected server node:

```
FORTE_ROOT/appdist/environment_ID
```

You may have to first create the *environment\_ID* directory, using the first eight characters of your deployment environment name, before you can copy the application distribution. On OpenVMS platforms, use the directory structure shown in the inset of [Figure 8-9 on page 273](#).

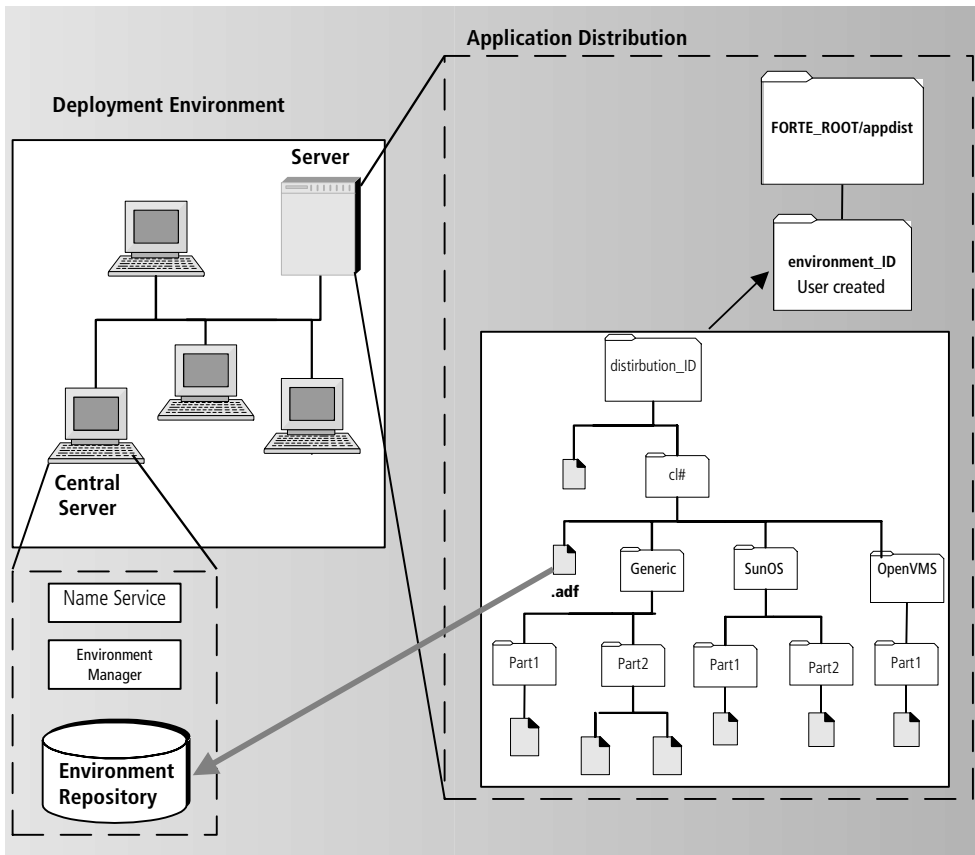
## Loading a Distribution into an Environment Repository

When you load a distribution, you are loading the information contained in the distribution's .adf file into your deployment environment's environment repository, as illustrated in [Figure 8-11](#).

The .adf file contains all the information needed by the Environment Manager to install all partitions onto the appropriate nodes in your deployment environment. For example, the .adf file contains instructions on where both standard and compiled partitions should be installed.



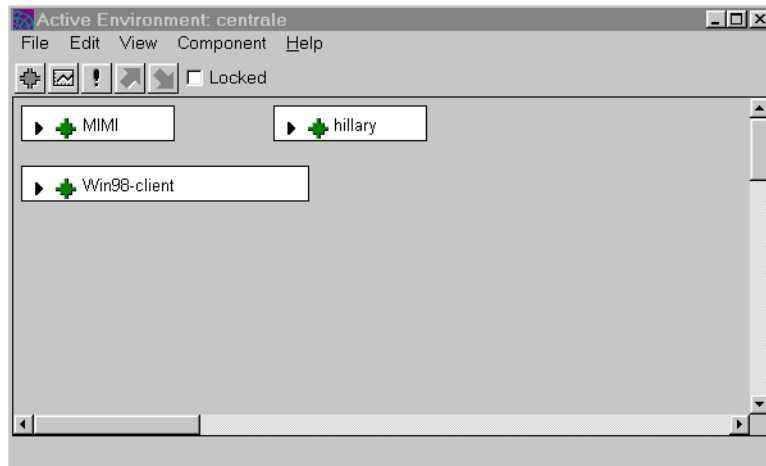
**Figure 8-11** Loading Distribution into Environment Repository



► **To load a distribution**

1. Make sure the distribution has been transferred to the deployment environment.
2. Open the Environment Console.

The Active Environment window appears.



In our example, the active environment, “centrale,” consists of three nodes: a model node for Windows NT clients, a Windows NT server node (MIMI), and a UNIX server node (hillary).

3. Select the File > Load Distribution command.

The Load Distribution dialog appears.



In the case of a local distribution, the window shows the FORTE\_ROOT/appdist directory for your local node. If you select a remote node from the drop list, the window shows the FORTE\_ROOT/appdist directory for the remote node. You can select a distribution to load from the displayed list.

In our example, the distribution resides on “hillary,” the central server node for the “DocEnv” environment. In the figure above, the TimeIt distribution is the only distribution on the node “hillary.” Within the TimeIt directory represented in the window, there is a single release of the TimeIt application distribution, “cl0,” indicating that this is the first release of the application (compatibility level is 0).

4. Select the distribution in the Load Distribution window, and click Load.

You can also load a distribution using Escript, as described in the *Escript and System Agent Reference Manual*.

## When a Distribution Conflicts with an Installed Application

If you are loading a new distribution for an application that is already installed in the environment, you might receive an error message that says that the application distribution you are loading conflicts with the installed application of the same name and release number. This message means that the configuration of the new distribution is different from the configuration of the installed application, which makes them incompatible.

Before you can load and reinstall the new distribution of the application, you need to uninstall the application currently installed in the environment.

iPlanet UDS determines whether two configurations of an application are compatible based on the following rules:

- For client applications and server applications, the new configuration must have the same number of partitions, and each partition must contain exactly the same service objects.
- For library distributions, the new release must have the same number of libraries and the names and UUIDs (universally unique identifiers) of each library must match exactly. The UUID of a library is taken from the UUID of its associated project. This UUID is assigned when the project is created in your repository.

To ensure that a given project has the same UUID, even when it is exported and imported, the application developer must export the project with their UUIDs. To export projects with UUIDs, use the Fscript **ExportPlan** command with the **ids** argument. For information about the **ExportPlan** command, see the *Fscript Reference Manual*.

## Modifying a Partitioning Configuration

After loading an application distribution, you can view and modify its partitioning configuration using the Environment Console or Escript utility. You would modify a configuration for either of two reasons:

- The simulated environment used by developers for partitioning the application may differ from your current deployment environment—a very likely case if you have made recent changes to your environment or the application was developed by an independent software vendor (ISV) or value added reseller (VAR) who had no knowledge of your particular environment.
- You anticipate performance or other problems in the partitioning configuration created by the application developers and therefore want to modify it.

You can modify any of the partitioning configuration properties normally specified in the right-hand panel of the iPlanet UDS Partition Workshop. (You *cannot* modify the logical partitioning scheme or change the replication properties of service objects.)

You can modify partition assignments and a number of assigned (installed) partition properties as described in the following sections.

### Reassigning Partition Assignments

You can reassign partitions to different nodes, make additional assignments of replicated partitions, or remove assignments. You can only reassign compiled partitions, however, to nodes corresponding to the architectures for which the partitions were compiled.

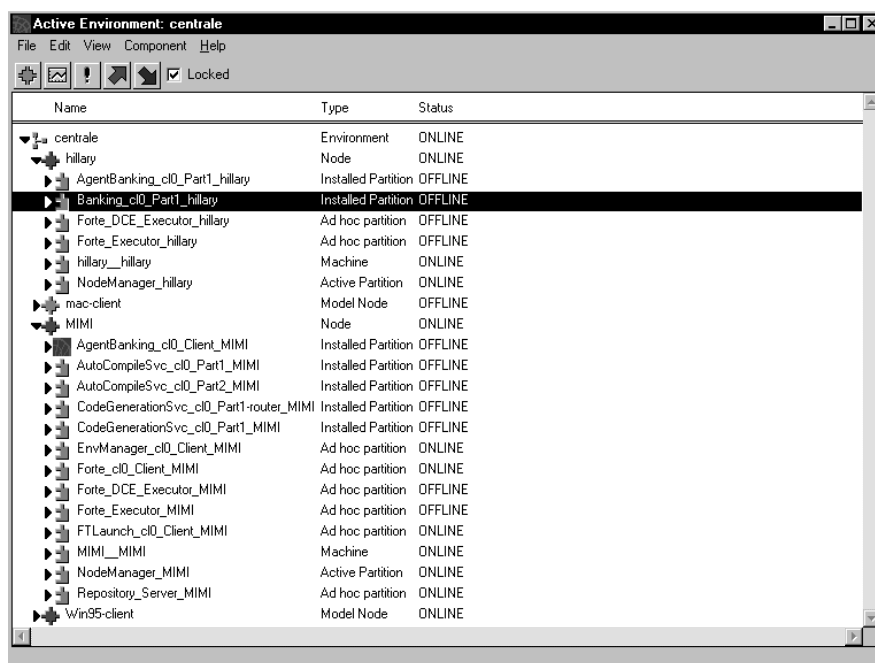
When an application distribution is loaded into your environment repository, iPlanet UDS checks the partitioning assignments. If any partition is assigned to a node not found in your environment, that assignment will be dropped. If all assignments for a logical partition are dropped, iPlanet UDS will perform a default configuration (just as it does in the Partition Workshop), assigning the partition to

a node in the environment. In this case, you may have to manually assign (or re-assign) the partition to a node (or nodes) in your environment. In assigning the partition, make sure that the target node has the platform architecture and resources required for the partition to run.

You cannot completely install an application unless every logical partition has been assigned to at least one node in your environment.

### ► To reassign a partition

1. Select the View > Node Outline view in the Active Environment window.
2. Expand the node to which the partition is currently assigned, as shown in the following figure:



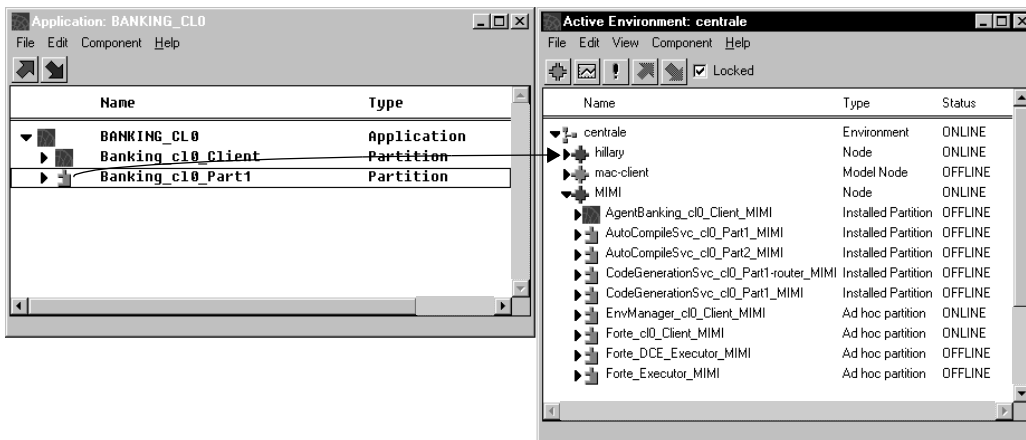
3. Drag the partition you wish to reassign and drop it on the node to which you wish to assign it.

➤ **To copy a partition assignment**

1. Select the View > Node Outline view in the Active Environment window.
2. Expand the node to which the partition is currently assigned.
3. Select the partition you wish to copy and copy it to the clipboard by choosing the Edit > Copy command.
4. Select the node to which you want to assign the partition and paste the partition from the clipboard by selecting the Edit > Paste command.

➤ **To assign an unassigned partition**

1. Select View > Application Outline in the Active Environment window.
2. Open the Application Agent window and expand the application in that window.



3. Select the View > Node Outline in the Active Environment window.
4. Drag the logical partition you wish to assign from the Application Agent window and drop it on the node to which you wish to assign it.

## Modifying Installed or Assigned Partition Properties

You can change some of the properties of partitions (described below) either:

- after you have loaded the application distribution, but before you have installed it

You can change these partition properties before you install the application.

- after you have installed the application

You can change these properties of the partition after you have installed the application. However, in this case, you need to reinstall the application after you change the partition's properties.

You can set the following properties for installed partitions:

**Compiled** You can turn off this toggle if you decide to run a partition in interpreted mode rather than as a compiled partition. You can also turn on this toggle to run a partition in compiled rather than interpreted mode, but only if the partition has already been compiled for the target node's architecture.

**Disabled** For replicated partitions, you can modify whether or not a partition is automatically started (enabled) on each node to which it is assigned by turning on or off this toggle. Every partition must be enabled on at least one node. If you disable a partition assigned to a node, it is not automatically started on that node.

**Thread Package** For server partitions only, you can specify whether the partition should run using DCE/POSIX threads or iPlanet UDS threads.

**Replication Count** For replicated partitions only, you can modify the number of replicates that are available for load balancing or failover.

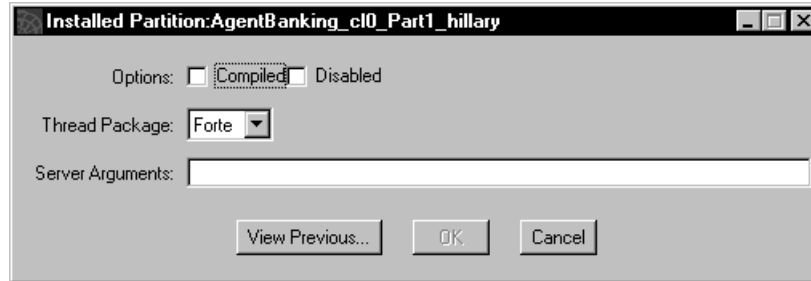
**Server Arguments** You can set flags that alter the default object memory space and logger settings set for a partition on startup.

The Installed Partition Properties dialog also displays the type of thread package that this partition uses. However, you cannot change this value in the Environment Console. You can only change the thread package specification for a partition in the Partition Workshop or in Fscript. For more information about thread packages, see *A Guide to the iPlanet UDS Workshops* and *iPlanet UDS System Installation Guide*.

### ► To set properties of an assigned or installed partition

1. In the Environment Console, choose the View > Node Outline command in the Active Environment window.
2. Expand the node to which the partition is currently assigned.

3. Select the partition and then choose the Component > Properties... command. The Installed Partition Properties dialog opens.



4. Change any of the fields described above and click OK.
5. Install the loaded application or reinstall the installed application that contains the partitions with the changed properties.

Before you reinstall an installed application to implement changes in its partition properties, you can compare the new property values for a partition with those for the currently installed partition. To see the properties for the currently installed partition, select the **View Previous...** button on the Installed Partition Properties dialog.

## Installing the Application

Once you have loaded your application distribution into the environment repository and made any changes in the partitioning configuration that you want, you are ready to install your application into your deployment environment.

Installation consists of downloading each partition to the node (or nodes) in your environment on which it is supposed to run. Installation is fully automated. The Environment Manager oversees and coordinates the downloading of partitions onto all nodes that have a Node Manager service running at the time of installation.

Throughout the application installation process, iPlanet UDS tracks the installation status of each node. It records the installation result for each targeted node in an environment, and posts events to announce the success or failure of installation on all targeted nodes.



If some target nodes are not available for installation—for example, server nodes that are not up and running or client nodes that are not currently running a Node Manager process—iPlanet UDS will perform a partial installation. Since iPlanet UDS keeps track of the installation status of each node, you can complete an installation incrementally as conditions permit, that is, as nodes become available for installation.

You can perform installations using either the Environment Console or Escript utility.

(In a development environment in which you wish to install an application *developed in that same environment*, you can install the application by selecting the Install in Current Environment option in the Make Distribution command, when making the application distribution.)

### ► To install an application

1. In the Active Environment window, select the View > Application Outline command.

In the Application Outline view, the top level is the list of applications, the second level is the set of logical partitions for the application, and the third level is the set of assigned nodes for the partition.

2. Select the application you want to install.

Notice that the Environment Console shows the status of the application as “Loaded.”

3. Select the Component > Install... command.

If your application successfully installs, the application status in the application outline view changes from “LOADED” to “OFFLINE.”

If the application status changes to “PARTIALLY INSTALLED,” the Environment Manager installed the application on as many nodes as were available, but not on all targeted nodes.

4. If necessary, complete a partial installation by starting Node Manager processes on the previously unavailable nodes and issue the Install... command.

On client-only nodes (Windows), use the Launch Server (or the Environment Console or Escript utility) to install the client partitions.

## Installing Applications on Server Nodes

For installation on server nodes, the Environment Manager informs each Node Manager of the application partitions that need to be installed on its node.

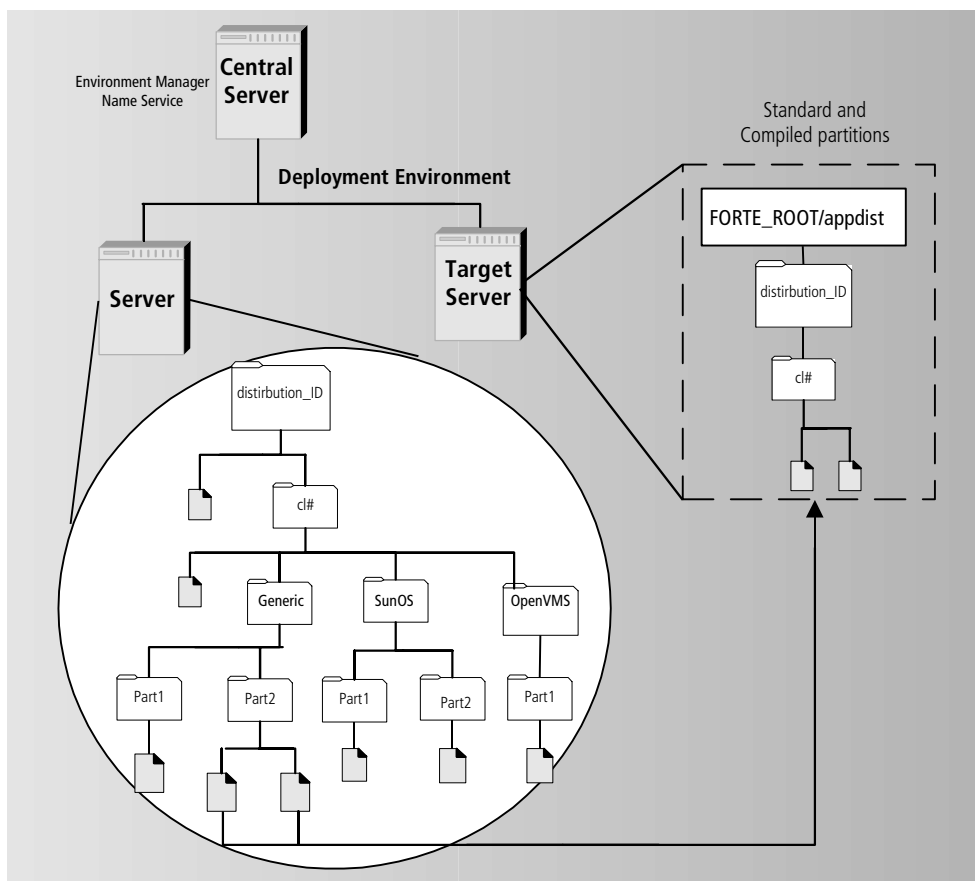
In the case of standard partitions, the Node Manager downloads the appropriate image repositories from the node on which the application distribution resides (if it does not reside locally). The image repository for a standard partition consists of two files named after the partition—one with an .btx and one with a .btd extension (distributions created before Release 3.0 are .idx and .dat files).

The Node Manager places each image repository in the following directory, as shown in [Figure 8-12](#):

**FORTE\_ROOT**/**userapp**/*distribution\_ID*/**cl#**

In the case of compiled partitions, the Node Manager downloads the appropriate executable partition files from the node on which the application distribution resides (if it does not reside locally) and also places them in the same directory as the standard partitions.

For the Node Manager to download partition files from a remote node, the Node Manager for the remote node must be up and running.

**Figure 8-12** Installing an Application on a Server Node

## Installing Applications on Client Nodes

Installation on client nodes is complicated by the fact that a client node, unlike a server node, does not usually run a Node Manager unless the client node normally starts a Launch Server.

If you want your application to be installed on client nodes as well as server nodes in your environment, start the Environment Console, the Escript utility, or the Launch Server on each client before you begin installation.

There are two ways of handling installations onto client nodes: explicitly and as-needed.

You can explicitly assign client partitions to specific client nodes or model node groups, as described in *“Reassigning Partition Assignments”* on page 292. You then start a Node Manager on the client node (by running the Environment Console, the Escript utility or the Launch Server). When you install the application using the Environment agent’s **Install** command, the Node Manager on the client node automatically installs the needed components on the client node. The Environment agent and its commands are described in the *Escript and System Agent Reference Manual*.

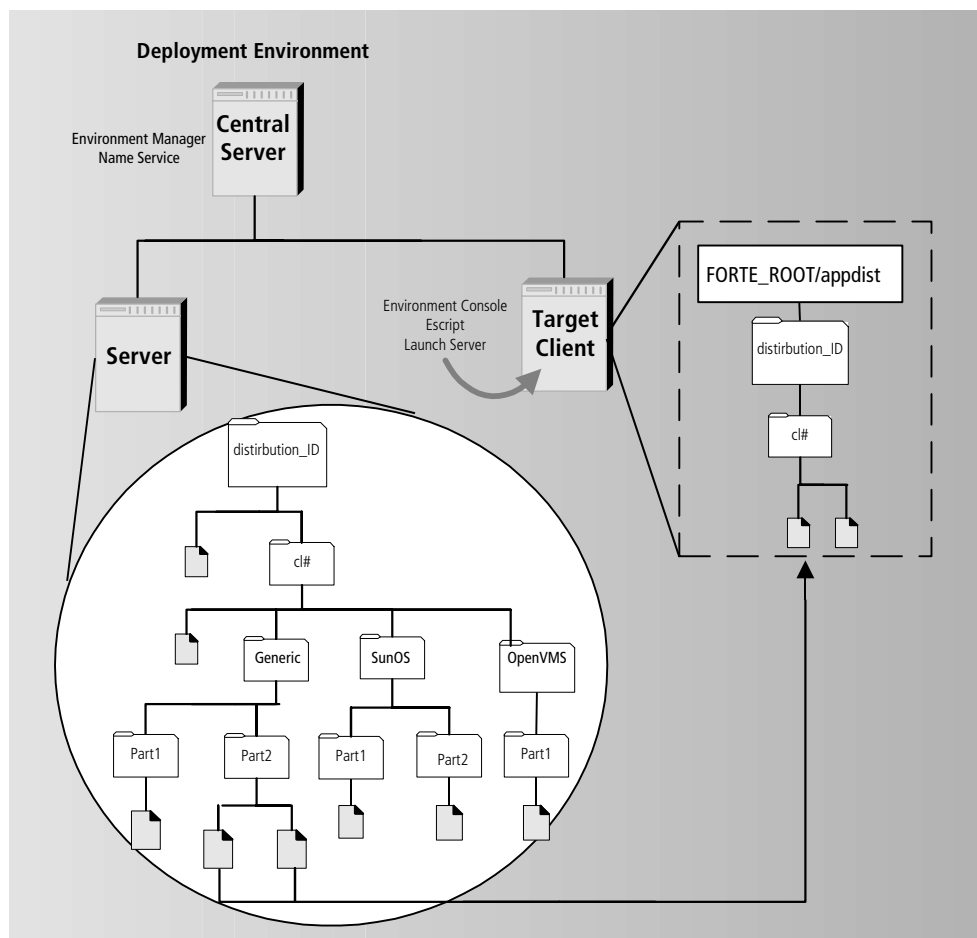
You can assign your client partitions or define them as publicly-available applications, and then let end users download the applications as-needed, when they want to run the application. This type of client installation works only for standard client partitions, not compiled client partitions. End users can also download updated releases of standard client partitions at runtime, if they use the Launcher application or if you set up their icons to do so. The Launch Server, along with the Launcher application and Ftcmd utility provide this download feature. This approach to installing client partitions is discussed in the *iPlanet UDS System Management Guide*.

In situations where it may not be practical to install your application on all client nodes simultaneously, you can perform the installation at a later time on a node by node basis.

The Node Manager service provided by these utilities downloads the files for the client partition from the application distribution, and places them in the following directory,

**FORTE\_ROOT/userapp/distributionID/cl#**

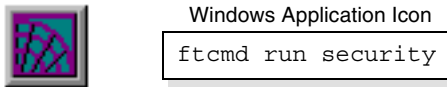
**Figure 8-13** shows the placement of the files.

**Figure 8-13** Installing an Application on a Client Node

### Generating Icons for Standard Client Partitions

In previous releases, iPlanet UDS generated icons for standard client partitions that start the application as a separate process in the client node's operating system.

iPlanet UDS now generates icons for standard client partitions that use the **ftcmd run** utility to start the applications.

**Figure 8-14** Windows Command Icon

iPlanet UDS generates icons on the Windows platforms that start the installed client partition using the Launch Server. These icons specify **ftcmd run** commands that have the Launch Server run the specified application. iPlanet UDS places these command icons in the iPlanet UDS program group.

On the UNIX platforms, you can define scripts or aliases that start client partitions using the **ftcmd run** command.

The Ftcmd utility is not available for OpenVMS platforms (VAX VMS).

The generated icons contain the **ftcmd run** command with the application name without the compatibility level.

You can also define icons that have the Launch Server start a client partition by using the following command syntax with the icons:

```
ftcmd run application_name [release] [arguments] [update]
```

For information about using this command, see the *iPlanet UDS System Management Guide*.

To have iPlanet UDS generate icons that start applications using the **ftexec** command, set the following configuration flag on each client node where you want this type of icon generated:

```
cfg:em:2
```

Set this configuration flag in either the Log Flags tab page of the iPlanet UDS Control Panel or the FORTE\_LOGGER\_SETUP environment variable. This configuration flag must be set before a Node Manager or a Launcher Server acting as Node Manager is started on the client node.

## Generating Icons for Compiled Client Partitions

If you are installing a compiled client partition, On Windows NT, iPlanet UDS creates an icon that starts the compiled client executable.

## Creating Icons by Hand

For those situations in which you may need to create an icon yourself to run an installed client partition, you can do so, as described below, by copying and modifying one of the existing iPlanet UDS system application icons (such as the Environment Console icon).

You can create icons that use a compiled executable, the **ftexec** command (described in the *iPlanet UDS System Management Guide*), or the **ftcmd run** command (described in the *iPlanet UDS System Management Guide*).

### ► To create a Windows NT client icon

1. Make a copy of the existing icon.
2. Select the icon copy, then open the Properties dialog using the Properties command on the popup menu.
3. On the Shortcut tab page, edit the Target field to provide the command-line argument to run your client partition.

## Installing Applications with Reference Partitions

Before you can install applications with reference partitions, you need to make sure that the application that reference partition points to is also installed.

If you are installing an application with a reference partition that points to a service that is only available in another environment, you need to make sure that the correct steps were followed for creating the replicated partition and setting the environment search path for this case. These steps are explained in detail in *A Guide to the iPlanet UDS Workshops*. You also need to make sure that this other environment is connected to your current environment. For information about connecting environments, see the *iPlanet UDS System Management Guide* and the *Escript and System Agent Reference Manual*.

## Completing Partial Installations

If any targeted nodes are unavailable at the time of installation, the installation of an application in your deployment environment will be only partially successful. A particular server node may be offline or a number of client nodes may not become available for installation until a later time. You might also add nodes to your environment and need to install partitions on these new nodes.

In such situations, the Environment Manager completes only a partial installation and reports which partitions await installation, and on which nodes. Using this information, you can complete partial installations as conditions permit.

You can complete a partial installation by re-installing the application when the needed nodes or Node Managers are up and running—the Environment Manager will attempt to complete what remains to be installed and issue a report.

## Deploying a Library

This section provides a description of how to define and deploy a simple library configuration using the iPlanet UDS Workshops and the Environment Console. For information about how to use Fscript and Escript commands to perform the same tasks, see the *Escript and System Agent Reference Manual* and the *Fscript Reference Manual*.

Before you can configure or deploy a library, you must be running the iPlanet UDS Workshops in distributed mode, which means you have clicked the iPlanet UDS Distributed icon or used the **forte** command without specifying the **-fs** flag. For more information about how to start the iPlanet UDS Workshops in distributed mode, see *A Guide to the iPlanet UDS Workshops*.

To deploy a library, you need to have write access to the volumes where you want to place the library distribution and where you want to install the library.

## Creating an Library Configuration

At this point, you should have written and tested the projects that you want to include in the library in the iPlanet UDS Workshops or Fscript.

You now need to define a configuration for your library so that you can install it in a deployment environment for testing and, eventually, customer use.

To create an application configuration, you need to define what partitions different parts of the application belong to. This task is described in [“Configuring Applications” on page 225](#) and is also called partitioning.

The simplest way to define a configuration is to let iPlanet UDS create a default configuration.



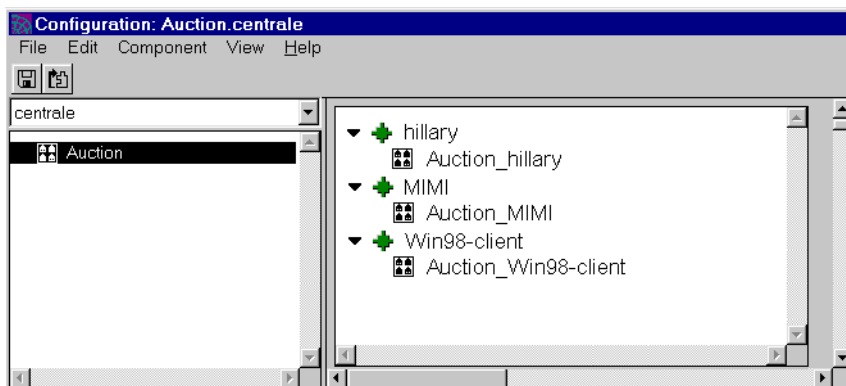
## Creating a Default Library Configuration

You can configure a library that contains one or more projects.

► **To create a default library configuration**

1. In the Project Workshop for one of the projects in the library, choose File > Configure as > Library.

The Partition Workshop opens, showing the project on the left and the default assignments of the library onto the nodes in the development environment, as shown below:



If you have already defined a custom configuration, but want to have iPlanet UDS replace that configuration with the default configuration, use the File > Repartition command.

The default configuration places the project you configured as a library on all nodes in the environment. When you modify the library configuration, you can add other projects to it and specify the nodes on which the libraries will be deployed.

Because the library configuration defines sets of libraries, not applications, you cannot run or debug it from the Partition Workshop, even if the project that was configured defines a start class and method.

## Examining Library Configurations

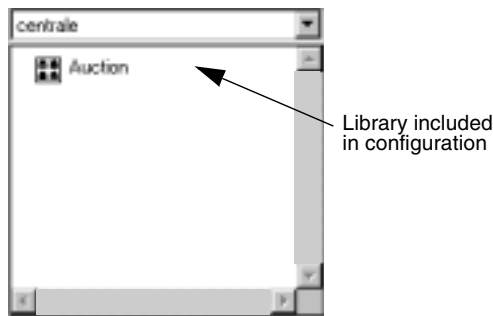
For library configurations, you can examine:

- the projects that are being configured as libraries
- the nodes in the environment
- the assigned libraries on specific nodes in the environment

### Examining the Projects

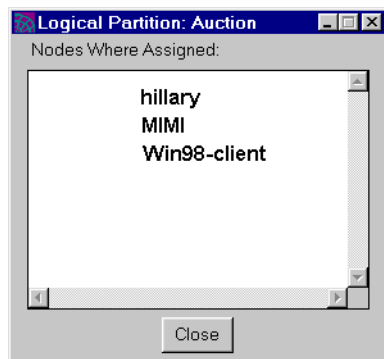
For library configurations, the Project browser portion of the Partition Workshop, shown in [Figure 8-15](#), lists the projects included in the library configuration.

**Figure 8-15** Project Browser



To view a list of the nodes to which the project is assigned, double-click on the project name. A dialog opens, as shown in [Figure 8-16](#), that lists the nodes where the project is assigned.

**Figure 8-16** Logical Partition Dialog for a Project



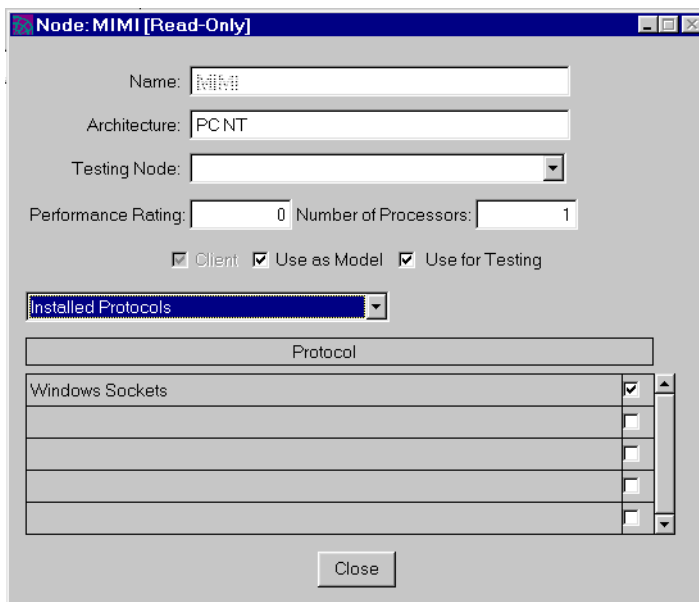
## Examining Nodes in a Library Configuration

For library configurations, the Nodes browser displays the assigned libraries within the environment. By default, the browser displays a topological view of the environment. The View > Node Outline command lets you display the same information in an outline form.

For each node in the environment, the browser displays a two-level hierarchy. At the top level is the node name. At the second level are the libraries assigned on that node. If the libraries are not currently displayed, open the node by clicking the expansion arrow.

To examine the node properties, double-click the node name, or select the node and give the Component > Properties... command. The Node dialog appears, shown in [Figure 8-17](#), displaying the node properties described under “[About Environments](#)” on page 234. These properties are the settings that were specified for the node when the environment was created in the Environment Console, and you cannot change them from the Partition Workshop. See the *iPlanet UDS System Management Guide* for further information on the properties of a node.

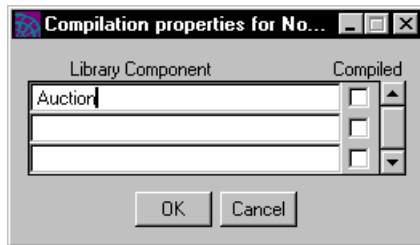
**Figure 8-17** Node Properties Dialog



## Examining Assigned Libraries

For each assigned library, you can open a dialog that shows which libraries on the node are compiled. To open the Compilation Properties for Node dialog, shown in [Figure 8-18](#), double-click the assigned library name or select the library and choose the Component > Properties... command. A toggle next to the library's name indicates that the library is compiled.

**Figure 8-18** Compilation Properties for Node Dialog



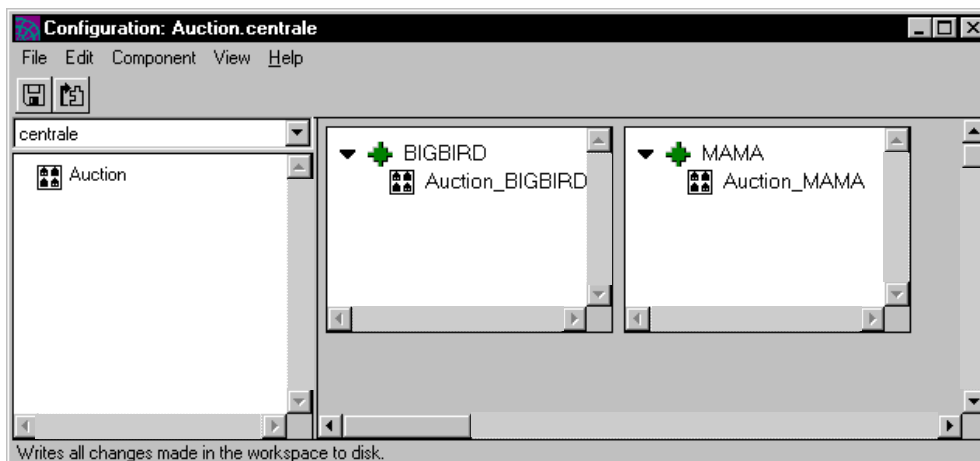
## Modifying a Library Configuration

Partitioning a library configuration is very different than partitioning a client or server configuration. When you partition a client or server configuration, you move partitions containing service objects onto nodes in the environment. When you partition a library configuration, you move *projects* onto nodes, not partitions.

A library configuration consists of projects assigned to nodes in the environment. Non-restricted projects can be assigned to any node in the environment. Restricted projects can be assigned to any node on which they will run.

When you first configure the project as a library, there is only one project in the shared library, the project that you have “partitioned” by giving the Configure as > Library command. This project you “partition” differs from other projects you later add to the library configuration only in that its name is used to name the library distribution and its compatibility level is used for the library distribution.

Otherwise, all projects in the configuration are exactly the same. [Figure 8-19](#) illustrates a default library configuration:

**Figure 8-19** Default Library Configuration

At this point, if you wish to package other projects in the library distribution, you can add them to the library configuration.

Note that to make permanent changes to a configuration, your workspace must be open for updating. If your workspace is open for reading only, you can use the Partition Workshop to examine a configuration and make temporary modifications to it. However, because you cannot save your workspace, any changes you make to the configuration are only temporary.

The following sections provide information about adding projects to the library configuration, deleting libraries from nodes, and turning on compilation for a library. Setting the configuration properties for a library configuration is the same as setting them for an application configuration. See [“Changing Configuration Properties” on page 268](#).

## Adding Projects to the Configuration

You can add any projects in your workspace to the library configuration.

When your library configuration contains more than one library, you must ensure that each library has a unique name. By default, iPlanet UDS uses the first eight characters of the project name as the library name. If two or more projects in the library configuration have names that start with the same first eight characters, you must specify unique library names for the projects. See *A Guide to the iPlanet UDS Workshops* for information on setting the library names.

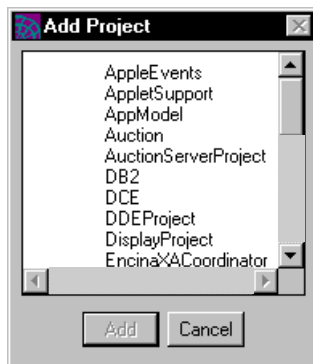
➤ **To add a project to the configuration**

1. Drag the project from the Repository Workshop to the Partition Workshop.
2. In the Partition Workshop, drop the project on any node where you wish to install it.

or

1. In the Partition Workshop, choose the Component > Add Project command.

The Add Project dialog opens.



2. Select the project you wish to add and click the Add button.

When you add a new project to the library configuration, iPlanet UDS automatically assigns the project to all the nodes where the original project is.

The following section describes how to modify this default configuration by removing projects from a node.

## Removing Libraries from a Node

You can modify a library configuration by:

- removing an individual restricted external library from a node
- removing all libraries from a node

Because restricted external projects cannot run on all nodes in the environment, you must remove them from the nodes where they cannot run.

➤ **To remove a restricted external library**

1. Select the assigned library.
2. Choose the Edit > Delete command.

3. Confirm that you wish to delete the assigned library.

If there is a node where you do not wish the libraries to be installed, you can remove them from the node by deleting any non-restricted project or restricted TOOL project. When you remove one of these projects from a node, all other projects on that node will also be removed.

## Standard or Compiled Libraries

By default, compilation for all libraries in the library configuration is turned off.

Remember, if a given library has supplier libraries, you must be sure to set the compilation options correctly for the main library and its suppliers. Once you designate a given library on a particular node as compiled, you must ensure that all its supplier libraries are also compiled. For a standard library, the supplier libraries can either be standard or compiled.

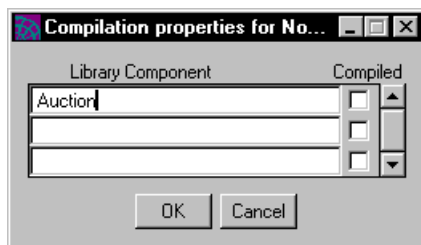
You also need to take into account whether or not the partition that will be using the library is compiled. If the partition is compiled, the library that it accesses must also be compiled. If the partition is standard, the library that it accesses can be either standard or compiled.

Furthermore, if any of the libraries will be used as suppliers to client partitions with C++ APIs, you need to compile the libraries and generate the handle classes for them. For information about integrating with C++, see *Integrating with External Systems*.

### ► To turn on compilation for a library

1. Double-click the assigned library.

The Compilation Properties for Node dialog opens.



2. Turn on the Compiled toggle next to the library name.

Remember, after specifying that the library is compiled, you may need to perform extra steps to produce the library distribution. See [“Compiling Libraries” on page 315](#) for information.

## Making a Library Distribution

This section describes how to create an iPlanet UDS library distribution for installation in iPlanet UDS deployment and/or development environments.

The process of making a library distribution is similar to making an application distribution. The major difference is that you can specify whether or not to include source code in the library distribution.

By default, source code is not included in the library distribution. When the library distribution is created without source code, iPlanet UDS developers using the library cannot view the source code for the methods, cursors, and event handlers in the library. A special option on the Make Distribution command allows you to request that source code be included in the distribution. When the library distribution is created with source code, iPlanet UDS developers using the library can view (but not modify) all the source code. (If you need to distribute source code that *can* be modified, you should simply ship the project export files.)

You create a library distribution from your library configuration, using the File > Make Distribution command in the Partition Workshop.

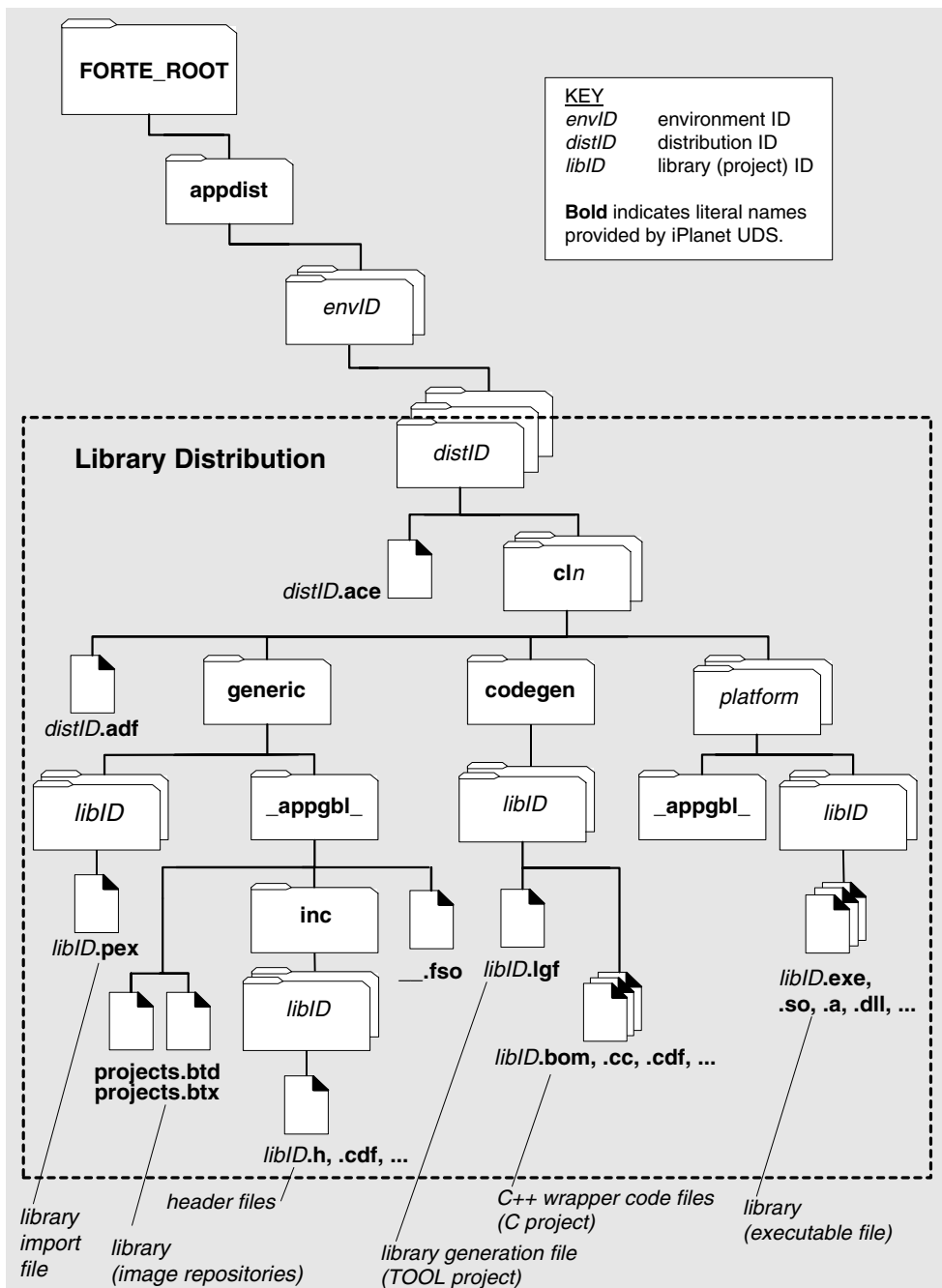
### About Library Distributions

A library distribution is a collection of files outside of the development repository that represent a library configuration (that is, a set of related shared libraries) intended for deployment. Once you create a library distribution, you can load the distribution into a target environment, and install it using iPlanet UDS system management tools.

[“Understanding Application Distributions” on page 270](#) describes the directory structure iPlanet UDS uses for an application distribution. The following figure shows the directories and files that iPlanet UDS adds to the distribution directory structure when you create a library distribution.



**Figure 8-20** Library Distribution Directory Structure



The following table describes these files.

Name	Purpose
—.ace	File which maps iPlanet UDS library configuration component names to unique identifier names.
—.adf	Application distribution file. Contains information about the library (partitioning) configuration—what libraries are assigned to what nodes—for use in the deployment process. (The .adf extension is added to the library configuration’s unique identifier name.)
generic	Directory that contains all portable files. The partition directories below the generic directory contain all the standard partitions.
codegen	Directory that contains all source files used for generating C++ compiled partitions.
platform1...	Directories that contain non-portable compiled partitions for each supported platform.
_appgbl_	Directory in which you or iPlanet UDS can place files that would be installed along with any library or set of libraries assigned to a node.
projects.btd, projects.btx	Standard image repository to be used on platforms for which no compiled libraries have been made.
—.fso	File that maps method names to ids, which might be used for some external interfaces or compiled partitions.
—.h, —.cdf, ...	Header files
—.pex	Export file used to import the library into development repositories in a target development environment.
—.lgf	Library generation file. Contains the source code used to make compiled libraries for a single project. (The .lgf extension is added to the library’s unique identifier name.)
—.bom, —.cc, —.cdf, ...	C++ wrapper files used as source code for making compiled libraries for a single C project.
—.so, —.exe, —.a, —.dll...	Compiled library file. (The name is platform dependent and based on library’s unique identifier name.)

## Using the Make Distribution Command

In the Partition Workshop, the File > Make Distribution command lets you make a distribution for a library configuration the same way you make an application. The only difference between using the Make Distribution command to create a library distribution is that you have the option of specifying that the source code be included in the library.

### ► To make a library distribution

1. Choose the File > Make Distribution command.
2. Select the appropriate distribution options on the Make Distribution dialog. The Include Source toggle lets you specify that the source code be included in the distribution.

When the Make Distribution completes, a message in the status line indicates that the distribution was complete.

If your configuration contains only standard libraries, or if your automatic library compilation completed successfully, you now have a library distribution, ready to install.

If your automatic compilation failed or if you did not select automatic compilation for your compiled libraries, you must compile the individual libraries before you can deploy the library distribution. The following section provides information about compiling libraries.

## Compiling Libraries

If your library configuration contains one or more compiled libraries and you did not select automatic compilation when you gave the Make Distribution command (or not all the libraries compiled successfully), you need to compile the libraries yourself using the **fcompile** command.

.When you specify that a library is “compiled,” the Make Distribution command creates a library generation file (.lgf), which you must use with the **fcompile** command to generate the compiled library.

If your library configuration contains multiple libraries, you must be sure to compile the libraries in the correct order. If any of the projects in the configuration are suppliers for other projects in the configuration, you must compile each of the supplier projects before compiling the main project.

Note that the automatic compilation feature of the Make Distribution command ensures that the libraries are compiled in the correct order. You need to be concerned about compilation order only when using the **fcompile** command.

You need to compile a library on the platform on which it will run. To do so, you move the .lgf file to the appropriate node, compile it, and then move the executable library file (and import file if appropriate) to the library subdirectory for the specific platform in the library distribution directory.

If your library configuration contains more than one library, you need to move and compile the libraries as a group, not one at a time.

► **To compile multiple libraries**

1. Move all the .lgf files to the nodes where they need to be compiled.
2. Compile all the libraries.
3. Move all executable library files and import files back to the appropriate subdirectories in the library distribution.

If the environment includes a mixture of platforms, you may need to compile a library on more than one platform. For example, let's say you are compiling a library for an environment that includes three servers: one server is an IBM RS/6000, one is a Sun SPARCStation running UNIX, and a third is a VAXStation running VMS. In this case, you would compile three separate executable library files using the same .lgf file.

Before running **fcompile**, check to see if you need to set environment variables prior to running **fcompile**. These environment variables are described under [“Environment Variables and Path” on page 279](#). You also need the C++ compiler. See the *iPlanet UDS System Installation Guide* for the currently supported compiler version.

### *Using the fcompile Command for Libraries*

The **fcompile** command generates code, compiles, and links a compiled library from an .lgf file. See [“Using the fcompile Command for Compiled Partitions” on page 281](#) for the complete syntax of the command.

When you give the **fcompile** command for a given library, you must do so on machine with the same architecture as the node to which the library is assigned. For example, if you are building a compiled library for an RS/6000, you must run **fcompile** on an RS/6000.

You cannot use a model node to create a distribution. Therefore, you should not run the **fcompile** command on a machine that is defined as a model node within the current environment.

► **To run `fcompile` for a compiled library**

1. If necessary, move the `.lgf` file to a machine that has the correct architecture.
2. Set the necessary environment variables, as described in “[Environment Variables and Path](#)” on page 279.
3. Run the `fcompile` command as described above.

```
fcompile -c aucio1.lgf -d ./rs6000
```

The `fcompile` command produces an executable library file in the format used by the specific platform. It may also produce a library import file.

4. Move the executable library file (and import file if appropriate) to the library subdirectory for the specific platform in the library distribution directory.

## Installing a Library Distribution

If your application requires access to libraries (shared libraries, object libraries, shared images, DLLs, or TOOL libraries), these libraries should be included in a library distribution packaged with your application distribution. The documentation accompanying your application distribution should include information about which libraries are needed by which partitions, so you can make sure that the appropriate libraries are installed on each node in your deployment environment.

iPlanet UDS automates the deployment of library distributions using its system management services. The process is essentially the same as for deploying application distributions: you load the library distribution into your environment repository, modify the library configuration to match your deployment environment and application partitioning configuration, and then install the library distribution into your deployment environment.

You can perform these tasks using either the Environment Console or the Escript utility. The steps for deploying library distributions using the Environment Console are summarized below. For information about using the Escript utility, see *Escript and System Agent Reference Manual*. Any differences between library distributions and application distributions are noted.

► **To deploy a library distribution**

1. Transfer the library distribution to a server node in your deployment environment.
2. Load the library distribution into your deployment environment's environment repository using the File > Load Distribution command in the Environment Console.
3. Modify the library's partitioning configuration, if you wish.

A non-restricted library is assigned, by default, to every server node in an environment. If your environment differs from the environment definition used to create the library configuration, you may have to assign the non-restricted libraries to nodes in your environment.

A restricted library requires special resources to support the library. You can assign a restricted library to nodes in your environment, or re-assign it, providing the target nodes have the resources to support the library.

Unlike partitions, libraries are not started by iPlanet UDS. Therefore, there are no properties analogous to the start options, server type, or server argument of assigned partitions. You can, however, choose whether to install each library as compiled or standard, if the distribution includes both the compiled executable and the image repository for the library.

4. Install the library configuration into your deployment environment.

iPlanet UDS automatically downloads each library to a standard location on the node (or nodes) in your environment to which it is assigned. Installed libraries, because they never become executing iPlanet UDS processes, have no corresponding system management agents—they are merely part of the definitional information stored in an environment definition.

Each node specification in an environment definition includes information regarding the restricted libraries installed on the node. In development environments, restricted library information is used in application partitioning.

In development environments, where developers need to access libraries to code and test their applications, you not only have to install the library configuration into the development environment, but you must also import the individual libraries into the development environment. Each library distribution includes a .pex file for each library that can be imported into development repositories, as shown in [Figure 8-20 on page 313](#).

# Removing and Updating Applications and Libraries

After you have deployed an application or library in a development or deployment environment, you will at some point need to remove or update the application or library. For example, you might need to replace an earlier release of an application with the latest release.

This section describes how to remove and update applications and libraries using the Environment Console. For information about performing the same tasks using Escript commands, see the *Escript and System Agent Reference Manual*.

## Removing an Application or Library

When you remove, or uninstall, an iPlanet UDS application or library, the Environment Manager deletes the information about the application from the environment repository and node repositories. After you uninstall an iPlanet UDS application or library, you can no longer manage the application or library using the Environment Console or Escript. However, the files for the application or library distributions still exist in the FORTE\_ROOT/userapp subdirectories where they were copied when the application or library was installed. If you want to delete these files completely, you can do so after you uninstall the application or library.

### ► To uninstall an application or library distribution

1. In the Environment Console, choose the View > Application Outline command to display a list of installed applications and libraries.
2. Select the Application agent for the application or library you want to uninstall. (Application agents also represent libraries.)
3. Choose the Component > Uninstall command to uninstall the application or library.

You can also use the Edit > Delete command to uninstall the application or library.

## Upgrading Applications

This section provides a brief explanation of how to perform the simplest upgrades of libraries and applications. For a more thorough discussion of the options you can consider when upgrading applications, including rolling upgrades, see [Chapter 13, “Upgrading Deployed Applications.”](#)

Although iPlanet UDS does not support versioning control, iPlanet UDS lets application developers differentiate between incompatible releases of an application by assigning each release a different compatibility level. Releases of an application with different compatibility levels are treated by the iPlanet UDS runtime system and by iPlanet UDS system management services as completely different applications.

For information about compatibility between releases, see *A Guide to the iPlanet UDS Workshops* and [Chapter 13, “Upgrading Deployed Applications.”](#)

Because these different applications can coexist in your deployment environment, you can deploy a new release of your application while the old one is still in use.

## Upgrading Installed Applications

By default, iPlanet UDS assumes that after you have installed an application, the configuration of the installed application is the correct configuration for the application. This assumption can affect whether you can install a new copy of the application over the installed application, or whether you need to uninstall the currently installed application first.

If you change the contents of any logical partition, iPlanet UDS cannot reinstall the application because the new configuration is incompatible with the configuration of the installed application. You need to uninstall the application before you can install a new distribution that uses a new configuration.

If you have changed only the partition assignments for an application that is already installed in the current environment, you can reinstall the application directly over the installed application distribution. However, the configuration of the application will remain the same as that for the previous installation of the application. The elements of the configuration that stay the same are the partition startup arguments, whether the partitions are compiled or interpreted, the number of replicates, where the partitions are assigned, and so forth.

If you want to change the configuration, you need to either change the configuration of the previous installation of the application before you install the newer application distribution, or you need to uninstall the application completely before installing the newer application distribution.



- **To change the configuration of an installed application without changing the contents of any logical partitions**
  1. Uninstall the application using the Environment Console or Escript. You might need to ask your system administrator to perform this task, depending on who is permitted to use the Environment Console or Escript in your environment. For information about how to uninstall an application, see [“Removing an Application or Library” on page 319](#).
  2. In the Partition Workshop, configure the application the way you want, then select the File > Make Distribution command.
  3. In the Make Distribution dialog, select the Full Make and auto-install options. You can also select the Auto Compile option, if appropriate. Click the Make button.

- **To upgrade an installed application**

1. Install the new release of the application on the clients and servers while the old release is running.
2. After all server nodes are running, move client nodes to the new release incrementally, by starting the new release on, for example, 50 of the clients, then 100, and so on until all clients are running the new release.

When you install an upgraded release of an application on your client nodes, be sure to upgrade the corresponding command icon, as well. For information about command icons, see [“Installing Applications on Client Nodes” on page 299](#).

If the application uses standard client partitions, you can have the end users upgrade their applications as needed using the services of the Launch Server, as described in the *iPlanet UDS System Management Guide*.

3. When all client nodes are running the new release, shut down and remove the old release from the server nodes and client nodes.

## Upgrading Reference Partitions

When you upgrade an application whose partitions are used as reference partitions for other applications, you need to make sure that the applications with reference partitions still reference the correct release of the partition. If the compatibility level of the application containing the partition has changed, then you need to reconfigure and redeploy the applications whose reference partition uses that partition, so that the reference partition uses the correct release of the application.

You do not need to change the compatibility level of the application containing the reference partition when you change the reference partition definition, and the compatibility levels of the two applications do not need to be the same.

For example, suppose you have an application `BankClient`, which contains a reference partition that represents a partition in the application `AppServices`. Both applications are at compatibility level 0. If you install a new release of `AppServices` at compatibility level 1, be aware that unless you update the reference partition in `BankClient`, that reference partition still references the partition of the older release of `AppServices`.

- ▶ **To make the reference partition reference the partition of the newer release of a changed application**
  1. In the Partition Workshop, view the configuration of the application with the reference partition and define a new reference partition that references the partition in the new release of the application. You can delete the old reference partition.
  2. Make a new distribution for the application with the reference partition.
  3. Reinstall the application with the reference partition in your deployment environment.

The steps for partitioning and making a distribution are described starting in [“Creating a Default Application Configuration” on page 240](#).

## Upgrading Libraries

When you upgrade a library to another compatibility level, you need to update and redeploy any applications that use this library. However, the compatibility levels of the library and its associated applications do *not* have to be the same.

If the compatibility level of the library does not change because the changes to the library are very minor, you do not need to upgrade the applications that use the library; you can simply install the revised library.

For example, suppose you have an application called `Payroll`, which uses classes and methods provided by the library `TimeCardFunctions`. Both the application and the library are at compatibility level 0.

If you install a new release of `TimeCardFunctions` at compatibility level 1, be aware that unless you update and redeploy the `Payroll` application, that application still references the older release of the `TimeCardFunctions` library.

► **To make the application reference the newer release of the library**

1. The application developer imports the .pex file from the library distribution into the development repository where the code for the application resides.
2. The application developer makes a new distribution of the application.
3. You install the updated release of the application in your deployment environment.

If you are upgrading libraries that are used by developers in a central repository, the developers need to check out all components of each library, reimport the .pex file for each upgraded library, then integrate the changed libraries into the system baseline. If the upgraded library is used in a private repository, the developer simply needs to reimport the new .pex file into the repository. For more information about using repositories, see *A Guide to the iPlanet UDS Workshops*.

## Partial Upgrades

When developers make an incompatible upgrade in an individual partition (or service object) in an application, they issue an upgraded release of the application—with a new compatibility level—for you to install in your deployment environment.

However, under certain circumstances, a new release of the service object in an application can be fully compatible with the rest of the application. In this situation, a developer can make a partial distribution without increasing the compatibility level of the application (see [Chapter 13, “Upgrading Deployed Applications”](#)). The partial distribution contains only the compatible, revised portions of the application.

You can deploy the partial distribution in your deployment environment just like any other application distribution. In this case, however, the distribution contains only the upgraded application partitions. When you deploy this partial upgrade, the new partitions are substituted for the old partitions. In this situation you must make sure that:

- the partitioning configuration of the upgrade distribution corresponds exactly to that of the old application
- your old application partitions are shut down before you install the new ones

For information about upgrading deployed and running applications, see the [Chapter 13, “Upgrading Deployed Applications.”](#)



# Class Runtime Properties

All iPlanet UDS classes have a set of characteristics called *runtime properties*. These properties determine how objects of each class are treated at runtime. Runtime properties can be allowed or disallowed for a class, or on by default for a class. The class runtime properties are the following:

- distributed
- shared
- transactional
- monitored

This chapter defines and describes each property and explains

- how you set a property for a class
- how you set or override attributes associated with the property for an individual object
- the effect of setting properties
- how deadlocks might occur and how to avoid such occurrences

## Class Runtime Properties

All iPlanet UDS classes have a set of characteristics called *class runtime properties*. When you create a new class, you can set one or more of these properties for the class. Class runtime properties define the runtime behavior of objects that are instances of each class. For example, if an object has the *shared* property, then access to that object is serialized so that multiple tasks do not attempt to update it simultaneously.

As you design or build an iPlanet UDS application, you should have a good understanding of how class runtime properties affect object behavior. The better your understanding, the more likely you can optimize your application's logic and performance.

The four class runtime properties are summarized below:

Property	Description
Distributed	Allows an object to be sent to a remote partition.
Shared	Allows multiple tasks to concurrently access and safely change an object's data.
Transactional	Allows an object to participate in a transaction.
Monitored	Indicates that an object may be displayed and mapped to a window widget.

Three of the class properties are associated with a corresponding attribute of the Object class (the Monitored property is not). The following table describes the Object attribute for each class property.

Class Property	Object Attribute	Description
Distributed	IsAnchored	<p>If set to TRUE, iPlanet UDS "anchors" the object to the partition in which it was created. When a remote partition references the object, a <i>distributed reference</i> is sent. Changes to the object are globally visible.</p> <p>If set to FALSE, the object is not anchored. When a remote partition references the object, a copy of the object is sent and all changes are made to the copy rather than the object.</p> <p>For more information see "<a href="#">Distributed Objects</a>" on <a href="#">page 330</a>.</p>
Shared	IsShared	<p>If set to TRUE, iPlanet UDS uses the locking necessary to synchronize access by multiple concurrent tasks. By default IsShared also sets IsAnchored.</p> <p>If set to FALSE, then an object should not be accessed simultaneously by multiple tasks.</p> <p>For more information, see "<a href="#">Shared Objects</a>" on <a href="#">page 335</a>.</p>

Class Property	Object Attribute	Description
Transactional	IsTransactional	<p>If set to TRUE, iPlanet UDS logs the state of the object before it is updated in a transaction, to allow changes to be undone if the transaction is unsuccessful.</p> <p>If set to FALSE, changes to the object are not logged, nor affected by the success or failure of a transaction.</p> <p>For more information, see <a href="#">“Transactional Objects” on page 343</a>.</p>

These attributes are also documented under the Object class in the Framework Library online Help.

For any given object to have a runtime property, either one of the following must be true:

- The property must be on by default for the object’s class.
- The object’s class must allow the property and the object itself must set the corresponding attribute.

For example, for an object called Balance to use transactional behavior, the transactional property must be allowed and the attribute IsTransactional set to TRUE. Or, if the transactional property is the default for a class, then the attribute IsTransactional is automatically set to TRUE.

## Class Runtime Property Defaults and Performance

Class runtime properties are predefined for all classes in iPlanet UDS libraries, such as Framework or Display; you cannot change the properties for these classes. However, you can override class properties for some subclasses.

When you create a new custom class, the default settings for all the class runtime properties are off (disallowed). To optimize the runtime performance of a class, in both a client and server, you should explicitly disable all class runtime properties. That is, you should set each property to Disallowed *and* untoggle the subclass override. This tells iPlanet UDS that the class and any of its subclasses requires no special handling and so can be manipulated very simply. For example, if Transactional is turned off, the system does not need to check whether it has to log an object before changing it.

Whenever possible you should turn off as many of the properties as possible without impacting the runtime behavior of your application. The next two sections describe setting class runtime properties.

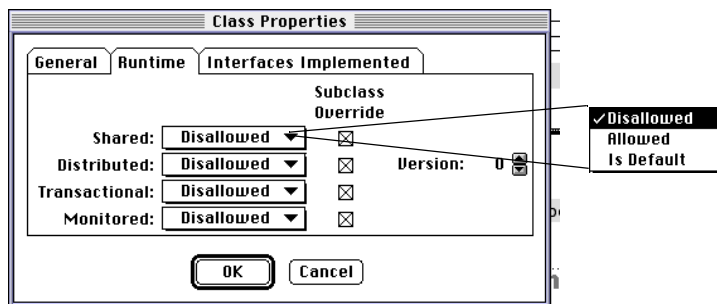
## Setting Runtime Properties for a Class

For a class that you have created, you can set the runtime properties at two levels:

- You can set the properties for a class as a whole.
- You can set the actual properties for an object of a class.

Using the Class Properties dialog in the Class Workshop, you can make a property Allowed, Disallowed, or the Default, as shown in [Figure 9-1](#).

**Figure 9-1** Class Properties Dialog: Runtime Properties Tab Page



If a class disallows a property, then no object of that class or its subclasses may take that property (unless Subclass Override is on; see below).

If a class allows a property, then any object of the class may take the property by setting the corresponding attribute. Thus, some objects of the class may have the property while other objects of the same class do not.

If a property is the default for a class, then all objects automatically have that property; the corresponding attribute is automatically set to TRUE. (One notable exception is that IsDefault properties are *not* applied to the starting class of a project; to get around this, you can assign class properties either in the Init method for the starting class or in the starting method for the project.)

When IsDefault is set for a class, the corresponding attribute can be explicitly set to FALSE to disable the property for an individual object.



If you turn on the Subclass Override toggle, then the property setting can be changed in subclasses of the current class.

For more information about using the Class Properties dialog, see *A Guide to the iPlanet UDS Workshops*.

## Setting Runtime Property Attributes for an Object

In some cases, you can explicitly set the properties for an individual object. Specifically, if a class definition allows the associated property, then you can set the `IsAnchored`, `IsShared` or `IsTransactional` attributes explicitly for an object of that class. Usually you will set these attributes in one of the following places:

- an object's constructor

For example:

```
data : SharedData = new(IsShared = TRUE);
```

- an object's Init method

For example:

```
method SharedData.Init()
begin
  self.IsShared = TRUE;
end method;
```

- the first statement after an object is allocated

The `IsDefault` runtime properties are set after the `Init` method completes.

If a class definition does not allow the associated property and you try to set the `IsShared` or `IsTransactional` attributes to `TRUE`, iPlanet UDS raises an exception.

## Runtime Attributes on Nested Objects

When you create a new object, the object runtime attributes are *not* set for nested object references. For example, if an anchored object has an attribute that should also be anchored, then you must also explicitly set the `IsAnchored` attribute for the nested reference. You can do this using the same mechanisms you used for setting `IsAnchored` on the outer object.

## Runtime Attributes on Cloned Objects

When you make a copy of an object using the Clone method, the current settings of the property attributes are *not* copied with the object, even if IsDefault is set for the property. Instead, they are all set to FALSE. If you want the copy to have any runtime attributes set to TRUE, you must explicitly set those attributes. For information on the Clone method, see the Object class in the Framework Library online Help.

# Distributed Objects

A *distributed object* is an object that has the following characteristics:

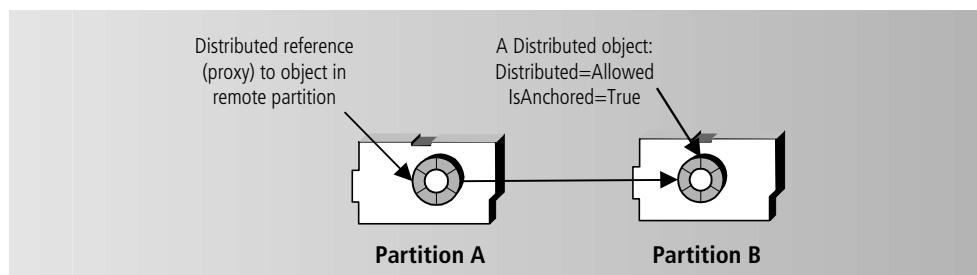
- it will be (or might be) referenced by multiple partitions
- as a global object, all changes made to it should be seen by all partitions

Note that iPlanet UDS service objects and shared objects are always (automatically) distributed objects.

You create a distributed object by allowing the Distributed property for the class in the Class Workshop and setting the object's IsAnchored attribute to TRUE. You can set the IsAnchored attribute to TRUE explicitly or by setting the default value for the class to TRUE in the Class Workshop.

If IsAnchored is TRUE, when the object is passed to another partition (through a method parameter, return value or event parameter), a *distributed reference* to the distributed object is actually passed (occasionally called a *proxy*). Every access to the distributed object from a remote partition uses a distributed reference. All changes are made in the partition on which the distributed object resides (requiring calls between the local and the remote partition) and all changes made to the object are visible to all partitions.

The relationship between distributed objects and distributed references is depicted in [Figure 9-2](#).

**Figure 9-2** Using Distributed References for Distributed Objects

If `IsAnchored` is `FALSE` (even if the `Distributed` property is allowed), then an object is *not* distributed. When the object is passed to another partition, a copy of the object is passed. Depending upon the method's parameter passing mechanism (for example, input, input/output, or output) the copy of the object might also be returned.

The interactions of the `Distributed` property and `IsAnchored` attribute are shown in the following table.

Distributed Property	IsAnchored Attribute	Effect
Allowed	FALSE	Object is not distributed, even though property is allowed.
	TRUE	Object is distributed.
Disallowed	FALSE	Object is not distributed.
	TRUE	Object is anchored, but not distributed. (See <a href="#">"Named and Unnamed Anchored Objects"</a> on page 332.)

When the `IsShared` attribute is set to `TRUE`, the `IsAnchored` attribute is automatically also set to `TRUE`, forcing all references to the shared object to refer to the same object. If the shared class will be passed to another partition, make sure to enable the `Distributed` property of the shared class.

## Named and Unnamed Anchored Objects

An anchored object can be either named or unnamed.

**Named anchored objects** These objects are explicitly registered with the iPlanet UDS name service, using the RegisterObject method on the ObjectLocationMgr class. Named anchored objects can serve a similar function as service objects. However, because they are named uniquely, they have an additional advantage in that they can be referenced selectively, unlike a service object. For more information about how you might want to use named anchored objects, see the ObjectLocationMgr class in the Framework Library online Help.

Like a service object, a named anchored object is given a dialog duration when it is registered with the ObjectLocationMgr. All three dialog durations are valid.

**Unnamed anchored objects** These objects are not registered with the iPlanet UDS name service. These objects always have a dialog duration of session.

## Non-Distributed Anchored Objects

Note that “anchored” is not synonymous with “distributed.”

Normally a distributed object has both the distributed property and the IsAnchored attribute set to TRUE. However, it is possible to set the IsAnchored attribute to TRUE for an object whose class definition does *not* allow the Distributed property. An object such as this is guaranteed never to leave its partition (thus the term *anchored object*).

Anchored objects are particularly useful when an object is platform-specific—for example, a file whose directory path is specific to one machine. If a coding error mistakenly passes a non-distributed, anchored object to another partition, an exception is raised. Thus, the IsAnchored property is useful to enforce desired application behavior.

## Invoking Methods on Distributed Objects

While there are some additional design considerations when you work with distributed objects, you need not adjust your code. The TOOL code is identical whether the object is remote or local, distributed or not; iPlanet UDS simply uses distributed references when acting on a remote object.

An object becomes distributed when a reference to it is passed to or made from another partition. The other partition locates the real object through the distributed reference. When a task in another partition invokes a method on the reference, iPlanet UDS applies a remote method invocation to the real object.

Remote method invocation is identical to local method invocation. If the method is invoked synchronously, then while the method is executing, the invoking task is “blocked” waiting for the method to complete. If the method is invoked asynchronously (using the **start task** statement), then while the method is executing, the invoking task is free to continue execution. You can use a return event to notify the calling task that the asynchronous method has completed.

## Accessing Attributes of Distributed Objects

Similarly, you can access attributes of a remote object just as any other object. iPlanet UDS wraps the remote attribute access like it was a remote method invocation. Attribute access is always synchronous.

When you access a remote object's attribute, you will get a distributed reference if the attribute is an anchored object (`IsAnchored=TRUE`). You will get a copy of the object if the attribute is an un-anchored object (`IsAnchored=FALSE`). If the attribute is a scalar, you get the scalar itself.

Consider the following distributed class `GlobalData` that has two attributes, a number and a nested object containing a number. An instance of the class (`TheData`) is marked as a service object:

### Code Example 9-1 Accessing attributes of a remote object

```
class NestedNumber inherits Object
  Number : Integer;
end class;
class GlobalData inherits Object
  Number : Integer;
  Nested : NestedNumber;
  has property
    distributed = (allow = on, default = on);
end class;
service TheData : GlobalData =
  (visibility = environment, dialogduration = session);
```

The following code fragment is executed in a different partition than where TheData is located:

```
old : integer = TheData.Number;  
TheData.Number = 0;
```

The first statement will retrieve the current value of the Number attribute and the second statement will set that attribute to zero. Now consider the following operation, which includes a reference to the nested object, Nested:

```
old : integer = TheData.Nested.Number;  
TheData.Nested.Number = 0;
```

The reference to TheData.Nested will request a local copy of Nested from TheData's partition. Thus, the first statement above will assign the correct value into the "old" variable. But the second statement will only replace the value in the local copy of TheData.Nested, not the intended value in the remote object's nested attribute.

You can avoid this error by wrapping attribute access in a method on the distributed class or by anchoring the nested objects. The following is a corrected definition for the NestedNumber class. The definition has added the Distributed property to make sure that all references to the object are through distributed references:

```
class NestedNumber inherits Object  
  Number : Integer;  
  has property  
    distributed = (allow = on, default = on);  
end class;
```

The following two statements are now correct because the reference `TheData.Nested` refers to an object anchored in the remote partition:

```
old : integer = TheData.Nested.Number;
TheData.Nested.Number = 0;
```

### Note on DataValue Subclasses in Framework

Some Framework library classes, such as `Array` and all the subclasses of `DataValue` (such as `IntegerData` and `TextData`), are *not* distributed and should not be anchored. You may set the `IsAnchored` attribute to `TRUE` for such a class, to force a runtime error to occur when you mistakenly attempt to send the object to a remote partition.

If you need to pass an object of one of these classes to another partition, you should make the object an attribute in another class that allows the `Distributed` property. Then, write a wrapper method to access the desired object. For example:

#### Code Example 9-2 Wrapper method to access classes that are not distributed

```
class DistributedName inherits Object
has private
  Text : TextData;
has public
  SetValue (Source: string);
  GetValue ():string;
has property
  distributed = (allow=on,default=on);
end
```

## Shared Objects

A *shared object* is an object that may be simultaneously accessed (possibly updated) by multiple tasks. Thus, access to the class must be controlled so that updates do not collide. Objects that are potentially manipulated by multiple tasks and are susceptible to access conflicts at runtime should be shared objects.

You create a shared object by allowing the `Shared` property for the class in the Class Workshop and setting the object's `IsShared` attribute to `TRUE`. You can set the `IsShared` attribute to `TRUE` explicitly or by setting the `IsDefault` value for the class.

When `IsShared` is set to `TRUE`, `IsAnchored` is also set to `TRUE`.

This section describes the interaction of mutex locks on shared objects. For a discussion of objects that are both shared and transactional, see [“Shared Transactional Objects and Transactional Locking” on page 347](#).

If you try to set the `IsShared` attribute to `TRUE` but the class definition does not allow the `Shared` property, an exception is raised.

## Automatic Locking: Mutexes

iPlanet UDS uses a locking mechanism called a *mutex* to prevent conflicts when multiple tasks try to access or change a shared object. If one task modifies a shared object's attribute, iPlanet UDS locks the object until the change is complete. If one task invokes a method on a shared object, iPlanet UDS does not allow another task to invoke a method on the object until the first task completes its method. Other tasks attempting to invoke methods on or access/modify attributes of the same object are “blocked.” Once the first task completes the method, another task is allowed to continue.

The order in which blocked tasks are allowed to continue execution is non-deterministic in the runtime environment. If a task locks a method and then terminates abnormally, the lock is released.

The following example illustrates when the lock is acquired and released in a method:

### Code Example 9-3 Using a mutex lock

```
i = sharedObj.Value;  -- Lock/unlock around attribute access
...
method SharedData.Update(i : integer)
begin
  ... -- Lock is held here
  if i < MAX then
    ...
    return; -- Lock is released upon leaving method
  end if;

  if errorOccurred then
    ... build exception ...
    raise errorMessage; -- Lock is released if we leave method
  end if;
  ...
  do other work;
end method; -- Lock is released upon leaving method
```



A mutex locks an entire object, including all its methods and attributes. A mutex cannot be established selectively for a method. A mutex incurs some overhead so you should set the `IsShared` attribute of an object only when necessary. For more selective locking, see the `Mutex` class in the Framework Library online Help.

A task started by another task has no special meaning for the purposes of locking shared objects. Even though they may be related in that they are part of the same “task family,” two tasks cannot concurrently lock a shared object’s mutex.

## Nonshared Objects and Concurrent Access

Generally speaking, multiple tasks should not operate concurrently on non-shared objects (objects whose `IsShared` attribute is not set to `TRUE`).

---

**CAUTION** If two tasks do operate on a single non-shared object, the results are unpredictable. It is possible that before one task completes the execution of one method, another task may begin executing the same or another method and may modify attributes being used by the first method. In a multithreaded server environment, the corruption of the object’s data may cause a server failure.

If `IsShared` is `FALSE`, you may still get concurrent access by explicitly using the `Mutex` class (described in the Framework Library online Help).

---

## Waiting for Events and Shared Objects

When a task is executing a method on a shared object, it owns the object’s lock. If, during the method, the task enters an **event loop** or **event case** statement, it gives up the lock before it waits for the events specified by the **event** statement. When the task gives up the lock, another task may invoke methods on the same object. In this manner it is possible for many tasks to register and wait for the same event on a shared object.

It does not matter if the events the task is waiting for are defined on the current object or not. Even if the task waits for the `Tick` event of the `Timer` class, it will still release the lock when it waits.

The following example illustrates when the lock is acquired and released within an event loop.

**Code Example 9-4** Using a lock in an event loop

```
method SharedData.Wait()
begin
  ... -- Lock is held here
  event loop
  ... -- Lock is still held here
  initialization code;
  ... -- Lock given up when about to wait
  when A do
    ... -- Lock reacquired upon event
    code to handle A;
    ... -- Lock given up when about to wait
  when B do
    ... -- Lock reacquired upon event
    code to handle B;
  exit;
end event;
... -- Lock is held here
end method; -- Lock is released upon leaving method
```

Subsequent posting of the event is broadcast to all tasks waiting for it. Once such an event is posted, the first task that receives the event will attempt to relock the shared object and execute the code block associated with the event. When this task returns to wait for more events or returns from the method, it will give up the lock, allowing the other tasks to execute on the object.

It is possible that while one task is waiting for an event from within a method being invoked on a shared object, another task may lock the object and invoke a method on it.

Note that it is important to wait for the event from within a method of the shared object's class. If you own the lock on a shared object and wait for an event in a method of another class, the shared object's lock will *not* be released while the task is waiting. You can also use this behavior as a mechanism to *avoid* releasing the lock when waiting for an event. Therefore, before writing an event loop for a shared class, you should confirm whether the default behavior is the one that fits your application.

## Nested Method Invocations

The task that holds the lock on a shared object is not blocked from invoking more methods on that object. The task may also invoke a method on a different object which, in turn, invokes a method back on the original shared object. This lock is granted because the task already owns the lock.

The following example illustrates the nested locking when the Update method invokes the MoreUpdate method. The “level” marks the nesting level of the lock:

### Code Example 9-5 Using a nested lock

```

data : SharedData = new(IsShared = TRUE);
data.Update();
...
method SharedData.Update()
begin
  ... -- Lock held at level 1
  self.MoreUpdate();
  ...
end method; -- Lock released at level 1 (unlocked)

method SharedData.MoreUpdate()
begin
  ... -- Lock held at level 2
  do some work;
end method; -- Lock released at level 2 (still locked)

```

## Common Mutex Deadlock

When multiple tasks access the same shared objects, tasks may occasionally block when trying to invoke a method or manipulate the attribute of a shared object. Usually the block is temporary, but in some situations, a *deadlock* may occur.

A *common mutex deadlock* occurs when two (or more) tasks each attempt to access the shared object that is locked by the other task. Neither task can move forward because it is waiting for the other task to release its lock. The following example illustrates a simple deadlock.

Two tasks, Task1 and Task2, are both working with two shared objects, Data1 and Data2. Task1 invokes `BeginUpdate` on Data1, thereby locking Data1; Task2 does the same with Data2. Then each task tries to lock the *other* object, invoking the `MoreUpdate` method. At this point, a deadlock error is raised. iPlanet UDS raises an exception for one task in order to allow the other task to proceed. Of course, the timing must be correct to force the deadlock condition to occur:

**Code Example 9-6** Example of a common mutex deadlock

```

Data1 : SharedData = new(IsShared = TRUE);
Data2 : SharedData = new(IsShared = TRUE);
start task Data1.BeginUpdate(Data2);
start task Data2.BeginUpdate(Data1);
...
method SharedData.BeginUpdate(other : SharedData)
begin
    ...
    other.MoreUpdate(); -- Possible deadlock
    ...
end method;

```

## Avoiding Common Mutex Deadlock

You can avoid common mutex deadlocks if you set the order in which shared objects are accessed. If all tasks lock the shared objects in the same sequence, this type of deadlock cannot occur. In the example above, if instead both tasks were to access Data1 first, then each task would be assured of obtaining the locks it needs without blocking the other task.

If an application cannot assume a particular access order, it should be prepared to handle the deadlock exception.

Note that transactional deadlock is always detected (see [“Transactional Deadlock” on page 351](#) for information).

## Distributed Mutex Deadlock

A task that currently has a mutex on one shared object may invoke a method on another shared object. When the other shared object resides in another partition the method invocation is remote. If an invocation order is not planned for remote method invocations on a shared object then a *distributed mutex deadlock* might occur.

iPlanet UDS cannot detect a distributed mutex deadlock at run-time.

An example of a distributed mutex deadlock follows, with two service objects, SO1 and SO2, and two clients, C1 and C2. C1 invokes the SO1.BeginUpdate method and C2 invokes the SO2.BeginUpdate method. With the right timing the two clients could end up deadlocked while waiting for mutexes in different partitions:

**Code Example 9-7** Example of a distributed mutex deadlock

```

service SO1 : SharedService;
service SO2 : SharedService;

SO1.BeginUpdate(SO2);    -- Client C1
...
SO2.BeginUpdate(SO1);    -- Client C2
...

method SharedService.BeginUpdate(other : SharedService)
begin
    ...
    other.MoreUpdate();  -- Possible distributed deadlock
    ...
end method;

```

You can avoid this type of deadlock by enforcing an order in which service objects are accessed. Consider carefully the locks you hold while invoking methods on a remote object.

## Distributed Recursive Deadlock

The task that holds the mutex lock on a shared object may invoke a method on another object, which, in turn, invokes a method on the original shared object. If the “other” object resides in the same partition as the shared object, the lock is granted. However, if the “other” object resides in another partition, then a *distributed recursive deadlock* condition occurs. In this type of deadlock, a task has propagated to a second partition, and while there has invoked a method on an object in the originating partition, thereby attempting to reacquire the lock which is held by the original caller.

A distributed recursive deadlock is not detected and will block indefinitely.

The following example shows two objects from two classes (SharedData and OtherData). The BeginUpdate method on SharedData invokes the CallBack method on OtherData, which invokes MoreUpdate back on SharedData. In the example, the “other” object is created in the same partition as the shared object. If, however, the other object resides in another partition, when that object tries to invoke MoreUpdate back in the shared object's partition, it will fail with a distributed deadlock error.

**Code Example 9-8** Distributed recursive deadlock

```

data : SharedData = new(IsShared = TRUE);
other : OtherData = new; -- In same partition so no deadlock
-- In remote partition deadlock error
data.BeginUpdate(other);
...
method SharedData.BeginUpdate(other : OtherData)
begin
    ...
    other.CallBack(self);
    ...
end method;

method OtherData.CallBack(sd : SharedData)
begin
    ...
    sd.MoreUpdate();
    ...
end method;

```

This type of deadlock is not avoided by ordering shared object (mutex) access. Instead, you must identify what remote methods you might invoke while locking the mutex and whether they, in turn, might invoke a method back on your object. A well designed object method interface with distinct layers to describe “who calls who” and “across what partition boundaries” can help eliminate this type of deadlock.

## Distributed Shared Objects

When the IsShared attribute is set to TRUE (explicitly or by default), the IsAnchored attribute is also set to TRUE; thus all shared objects are also distributed. Thus, a shared object never leaves the partition in which it was created; instead iPlanet UDS actually passes a distributed reference whenever that object is passed between partitions.

If iPlanet UDS attempts to pass a shared object to another partition, but the Distributed property for the object's class has not been set to Allowed, then you will get an exception. In some applications you may prefer to get the exception (rather than set the property to Allowed), because the exception may alert you to a particular condition, such as an object that is private to the partition being referenced remotely.

## Cloning Shared Objects

When you clone a shared object, the resulting clone is not shared. The IsShared attribute is not set even if the Shared class property default is set to on. Cloning a shared object is useful to save the object's state at a particular time. If you want to make the clone shared, you must set the IsShared attribute after the clone, as in the following example:

**Code Example 9-9** Marking a cloned object as a shared object

```
old : SharedData = current.Clone(deep);
old.IsShared = TRUE;
method SharedData.Init()
begin
    self.IsShared = TRUE;
end method;
```

## Transactional Objects

A *transactional object* is an object that can participate in a transaction. Changes made to the object during a transaction are logged so that they can be rolled back if the transaction is aborted. Objects that are both shared and transactional require *transactional locks* in addition to the mutex locks required by non-transactional shared objects. Outside the context of a transaction, a transactional object is exactly like any other object.

While a transaction is active, the transactional objects affected by the transaction may be locked, and remain so until the transaction ends. You should consider the duration of transactions as you design applications, particularly applications for a high number of concurrent users. If possible, you might restructure transactions to reduce the length of time locks are held.

You create a transactional object by allowing the Transactional property for the class in the Class Workshop, and by setting the object's IsTransactional attribute to TRUE (explicitly or by default). If you try to set the IsTransactional attribute to TRUE, but the class definition does not allow the Transactional property, an exception is raised. You do not need to be in a transaction in order to set the IsTransactional attribute.

Transactional objects may reference other transactional and non-transactional objects. If you set the IsTransactional attribute to TRUE for an object, this does *not* automatically set the IsTransactional attribute for objects it references. You must explicitly set the attribute for any nested references. This is illustrated in the following example in which the Init method sets the IsTransactional attribute for nested object references:

**Code Example 9-10** Making nested attributes transactional

```
class DataSet inherits Object
  Number : Integer;
  Text   : TextData;
  List   : Array of TextData;
  Init();
  has property
    transactional = (allow = on, default = on);
end class;

method DataSet.Init()
begin
  Text = new(IsTransactional = TRUE);
  List = new(IsTransactional = TRUE);
end method;
```

For convenience, some Framework classes, such as Array and all the subclasses of DataValue (such as IntegerData and TextData), allow the Transactional property. You may set the IsTransactional attribute of these types of objects at runtime, as in the example above.

However, most iPlanet UDS classes are not transactional. For example, if you write to a File object during a transaction, the contents of the file do not get rolled back if the transaction is aborted. See the descriptions of individual classes in the iPlanet UDS documentation set to see which classes are transactional.



## Transactional Logging

If you modify the attributes of a transactional object during a transaction, the object's initial value is logged before the first modification. If the transaction is aborted, the log is used to restore the object to its initial value.

The attributes of non-transactional objects may also be modified during a transaction, but those changes are not rolled back if the transaction aborts.

iPlanet UDS variables are not transactional; their values are never rolled back. For example, local scalar variables in a method (such as integer, string, and boolean) are never transactional. If you change the value of a scalar variable during a transaction and the transaction is aborted, the value of the variable is not rolled back. The object that a variable points to may be transactional, but the value of the variable (that is, the object the variable points to) is not transactional.

The following example illustrates the transactional behavior of three local variables. The Data class is transactional (without a default). Three variables exist (two objects and one scalar), only one of which is logged:

### Code Example 9-11 Effect of transactional logging

```
class Data inherits Object
  Value : String;
  has property
    transactional = (allow = on); -- No default setting
end class;

...
tran    : Data = new(IsTransactional = TRUE); --Transactional
nonTran : Data = new; -- Non-transactional
scalar  : string; -- Scalar variable

...
tran.Value    = 'before-1';
nonTran.Value = 'before-2';
scalar        = 'before-3';

begin transaction
  tran.Value    = 'after-1'; -- Logged before change
  nonTran.Value = 'after-2'; -- Non-transactional - not logged
  scalar        = 'after-3'; -- Scalar variable - not logged
  ...
  transaction.Abort(TRUE);

  exception
  ...
end transaction;
```

The values after the transaction is aborted are only restored for the transactional object, tran:

```
tran.Value      : 'before-1'; -- Rolled back
nonTran.Value  : 'after-2';
scalar         : 'after-3';
```

An event posted during a transaction, even if posted on transactional objects, is not transactional and is not logged. If you post an event on a transactional object and then abort the transaction, the event remains posted. You should pay special attention if you post an event during a transaction that contains a parameter that is a copy of the current state of a transactional object. If the transaction is aborted, the value of that event parameter is inconsistent.

One useful technique may be to post a distributed reference to the transactional object and not a copy. Another useful technique is to post a commit-like event that allows event recipients to be notified that their data is now committed, after which they may retrieve the data's value.

## Common Transactional Logging Error

A common error is when a transactional object references a non-transactional object whose changes are not rolled back after the transaction is aborted. In the following example, the Data class definition includes a single attribute Text that is not transactional:

### Code Example 9-12 Common transactional logging error

```
class Data inherits Object
  Text : TextData;
  Init();
  has property
    transactional = (allow = on, default = on);
end class;

method Data.Init()
begin
  Text = new;          -- No setting of IsTransactional
end method;
```

Then the value of the Text object is modified during a transaction, which is then aborted:

```
d : Data = new;
d.Text.Value = 'before';
...
begin transaction
  d.Text.Value = 'after';
  transaction.Abort(TRUE);
...
end transaction;
```

After aborting the transaction, the value of `d.Text` is “after” because the changes were *not* rolled back. To have the Text object participate in the transaction, you could simply modify the `Init` method to make the Text object transactional, as follows:

```
method Data.Init()
begin
  Text = new(IsTransactional = TRUE);
end method;
```

## Shared Transactional Objects and Transactional Locking

A transactional object can also be shared; that is, both the `IsTransactional` and `IsShared` attributes are set to `TRUE`. In order to access or modify a shared transactional object (either through an attribute or a method), the transaction must acquire a *transactional lock* on the object. In fact, one transaction could easily have transactional locks on a number of transactional objects.

The *transactional lock* is in addition to the normal *mutex lock* that regulates concurrent shared object access. A mutex lock and transactional lock differ in two key ways:

1. A mutex lock is held by the *task* for the scope of the method or attribute access.

A transactional lock is held by the *transaction* for the duration of the transaction. A transactional lock is not temporarily released when the task in the transaction waits for an event on the shared transactional object.

2. A mutex lock is always acquired exclusively. Tasks do not share a single mutex lock.

A transactional lock can be acquired in either read (shared) mode or write (exclusive) mode. iPlanet UDS automatically determines the mode of the transactional lock required when accessing the shared transactional objects.

The fact that a transactional lock is required in addition to the mutex lock does not disable or modify any of the policies iPlanet UDS applies with the mutex lock.

As a general rule, iPlanet UDS guarantees that each shared transactional object can support either:

- one or more transactional read locks
- one transactional write lock

When a transaction attempts to access a shared transactional object, iPlanet UDS decides if the transactional lock may be granted to the transaction. If no other transaction is holding the lock, then the access is granted. If another transaction is holding the lock in a compatible mode (that is, both are “reading” a value from the object and require a read lock), then the lock is granted. If another transaction is holding the lock in an incompatible mode (that is, the other transaction previously modified the contents of the object and holds a write lock), then the lock is denied and the request is blocked. If blocked, the transaction waits until the lock is released.

Once a transaction is granted a lock on an object, it holds it until the transaction ends (either by aborting or committing). If an exception occurs that aborts the transaction (or cancels the task), it is treated like the transaction aborted. When the transaction ends, the lock is released and the next waiting transaction (if any) is granted access.

When your application requires multiple independent transactions that might be executed concurrently on a set of shared objects, take the locking semantics into consideration. A simple method invocation may block for a long period of time until an interactive user clicks a Save button, which commits a transaction and releases the transactional lock that the method invocation is waiting for.

## Updating Non-Shared Transactional Objects

Transactional locks prevent multiple independent transactions from simultaneously modifying the attributes of a *shared* transactional object (by serializing the updates).

However, if multiple independent transactions modify a *non-shared* transactional object no locking mechanism exists to serialize the updates; iPlanet UDS does not enforce locking for non-shared objects. Thus, for non-shared objects, one transaction might update the object and, before that transaction has ended, another transaction might modify the object again, invalidating the first transaction's change.

---

**CAUTION** To avoid data corruption you should mark as shared any transactional object that could potentially be modified by multiple concurrent independent transactions.

---

## Read Locks

A *read lock* is a non-exclusive (shared) lock that can be held by multiple transactions. A read lock is required when a task accesses the value of an attribute on a shared transactional object, or when it invokes a method that *potentially* does not modify any of the object's attributes. Another task executing in a different transaction may access the same attributes and methods (that do not modify the object) and obtain a read lock, because iPlanet UDS ensures the transactional integrity of the data.

However, a read lock cannot be held at the same time as a write lock. If a transaction holds a read lock, it blocks a write lock request from another transaction. Or, a read lock that is requested while another transaction holds a write lock is blocked until that transaction ends.

The following example shows an attribute access and a method that does not modify any of the object's attributes. If the statement or method is executed by a task in a transaction, iPlanet UDS automatically requests a read lock for the object:

### Code Example 9-13 Transactional read lock

```

val = sharedTranObj.Value; -- Read lock before access
...
method SharedTran.GetTotal() : double
begin -- Read lock requested here
  sum : double = 0.0;
  for a in self.Accounts do -- Read self.Accounts
    sum = sum + a.Balance;
  end for;
  return sum;
end method;

```

## Write Locks

A *write lock* is an exclusive lock on an object; multiple transactions cannot concurrently hold any locks on an object while one transaction holds a write lock on the object. Another task in a separate transaction that is trying to access or assign to the object is blocked until the task that holds the write lock ends its transaction.

A write lock is required when a task modifies the value of an attribute on a shared transactional object or when it invokes a method that *potentially* modifies any of an object's attributes. A write lock that is requested while multiple transactions hold a read lock is blocked until all those transactions have ended.

The following example shows an attribute assignment and a method that might modify one of the object's attributes. If the statement or method is executed by a task in a transaction, iPlanet UDS will automatically request a write lock for the object:

### Code Example 9-14 Transactional write lock

```
sharedTranObj.Value = val; -- Write lock before assignment
...
method SharedTran.GetCurrent() : integer
begin -- Write lock requested here
  for a in self.Accounts do
    if a.Expired then
      self.Current = self.Current - 1; -- Write self.Current
    end if;
  end for;
  return self.Current;
end method;
```

Even though the `GetCurrent` method may never execute the code that modifies the `Current` attribute, iPlanet UDS requires a write lock in order to execute the method.

In the following code line, `Outer` and `Nested` are shared transactional objects. If the statement is executed in a transaction, iPlanet UDS requests a read lock for `Outer` (accessing the `Nested` attribute) and a write lock for `Nested` (assigning to the `Value` attribute):

```
Outer.Nested.Value = 'new value';
```

## Lock Promotion

When a transaction holds a read lock on a shared transactional object, it may execute some code that requires a write lock on the same object. Converting a lock from a read lock to a write lock is called *lock promotion*.

A transaction may block when promoting a lock. For example, assume two transactions share a read lock and one transaction requests a write lock (promoting the lock). The transaction requesting the write lock must wait until its partner in the read lock ends its transaction and releases the lock, allowing the promotion to continue.

## Transactional Deadlock

When multiple transactions access and lock the same shared transactional objects, a transaction may occasionally block when trying to invoke a method or manipulate the attribute of such an object. In some cases, a *transactional deadlock* may occur. Although similar to mutex deadlocks (see [“Common Mutex Deadlock” on page 339](#)), transactional deadlocks are more common than mutex deadlocks. Transactional locks are typically longer in duration, with a higher probability of blocking another transaction. Unlike mutex deadlocks which can go undetected by iPlanet UDS under some circumstances, transactional deadlocks are always detected by iPlanet UDS.

Deadlock occurs when two (or more) transactions attempt to manipulate a shared transactional object that is locked by the other transaction. Both transactions cannot move forward until the other transaction ends, thus releasing its lock.

The following example illustrates a simple transactional deadlock condition. The two transaction blocks, T1 and T2, are executed in parallel by two tasks. Transaction T1 locks object o1 and then transaction T2 locks object o2. Both locks are exclusive write locks. Now T1 tries to lock o2 and blocks, waiting for T2. When T2 continues and tries to lock o1, a transactional deadlock error is raised:

### Code Example 9-15 Transactional deadlock

```
T1: begin transaction
o1.Value = 1;  -- Lock o1
o2.Value = 2;  -- Wait for T2
end transaction;
```

```
T2: begin transaction
o2.Value = 22;  -- Lock o2
o1.Value = 11;  -- Wait for T1 !
end transaction;
```

You can use the same technique to avoid transactional deadlocks as to avoid mutex deadlocks. See [“Avoiding Common Mutex Deadlock” on page 340](#). You must be able to define one sequence in which all transactions will access the shared transactional objects. If you cannot define such a sequence because objects must be accessed randomly, then your application should have provisions for transactional deadlocks.

## Lock Promotion Deadlock

Another form of transactional deadlock is the *lock promotion deadlock*. In this case, multiple independent transactions have already been granted a read lock to the same shared transactional object. If one of the transactions next attempts to modify the object and now requires a write lock, it will block pending the end of the other transactions that are holding the read lock. If, however, another of the transactions also attempts to modify the object, this will cause a lock promotion deadlock.

The following example illustrates this condition. The Artist class is transactional, and it has Name, Comments, and Country attributes. The following code is part of a UserWindow method that references Artist:

**Code Example 9-16** Lock promotion deadlock

```
-- Object 'a' of type Artist is passed in, and shared.
begin transaction
  event loop
    when <read_lock>.click do
      t : TextData = a.name; --Read lock
      log.Putline(t);
    when <write_lock>.click do
      a.name = new(value = 'New Value'); -- Write lock
      log.Putline(a.name);
    when <commit_xact>.Click do
      exit;
  end event;
exception
  when e : GenericException do
    task.ErrorMgr.ShowErrors();
end transaction;
```

If you start up two separate windows, click on the read button, and then the write button, you will get a deadlock exception. The deadlock exception occurs because the read operations get read access to the object, and then the write operations attempt to promote the read access to write access.



The workaround is simple; avoid the lock promotion by acquiring a write lock initially instead of a read lock. In the above example, you could substitute the following code:

```
when <read_lock>.Click do
  a.name = a.name; -- Gets a write lock.
  t : TextData = a.name; -- Already has write lock
  log.Putline(a.name);
```

This code causes the second attempt at the read operation to block until the first transaction is committed, and the deadlock does not occur.

Another example of lock promotion deadlock follows. When transaction T1 tries to convert the lock on object “o” from a read lock to a write lock, it blocks. If transaction T2 tries the same operation, it will block waiting for T1 and a transactional lock conversion deadlock error is raised:

**Code Example 9-17** Two transactions in lock promotion deadlock

```
T1: begin transaction
  v = o.Value; -- Lock o (read)
  o.Value = 1; -- Convert/wait
end transaction;
```

```
T2: begin transaction
  v = o.Value; -- Lock o (read)
  o.Value = 2; -- Convert/Wait!
end transaction;
```

This case may seem unusual, but, with the flexibility of distributed objects, this can occur quite simply. The class ServiceObject below has two methods: GetId (which does not modify any attributes) and SetUser (which sets the current user name). A service object SO is created for the class:

```
class ServiceObject inherits Object
  has private
    Id : integer;
    User : TextData;
    GetId() : integer;
    SetUser(user : TextData);
  has property
```

```

        transactional = (allow = on, default = on);
        shared = (allow = on, default = on);
    end class;

    method ServiceObject.GetId() : integer
    begin
        return self.Id;
    end method;

    method ServiceObject.SetUser(user : TextData)
    begin
        self.User = user;
    end method;

    service SO : ServiceObject =
        (visibility = environment, dialogduration = session);

```

At some initial point in the application, client code executes the following statements:

```

begin transaction
    ...
    id = SO.GetId();           -- Read lock on SO
    ...
    busy work;
    ...
    SO.SetUser(myName);       -- Write lock on SO
    ...
end transaction;

```

Upon the invocation of `GetId`, the caller has a read lock on the service object. Then, when `SetUser` is invoked, a write lock is requested. Imagine that while one client is doing the “busy work,” another client starts and also invokes the `GetId` method, thus sharing the read lock. If they continue executing, they will encounter a lock promotion deadlock exception.

You can avoid lock promotion deadlocks in more than one way.

One way is to change any method on a shared transactional service object that modifies an attribute to include a statement that requires a write lock. For example, the `GetId` method could be changed as follows:

```
method ServiceObject.GetId() : integer
begin
    self.Id = self.Id;
    return self.Id;
end method;
```

Another way is to provide a method that guarantees a write lock request and define the class interface to call this method first. For example, if the interface to the `ServiceObject` class above required that you first invoke `SetUser`, then a write lock could be requested at the start of the transaction.

## Locking in Nested Transactions

Exclusive locks acquired in nested transactions are released if the nested transaction is aborted. If the nested transaction commits, then ownership of the exclusive lock is passed up to the enclosing transaction and held until that transaction ends.

Shared locks acquired in a nested transaction are treated as though they were granted to the enclosing transaction.

## Transaction Task Participants and Locking

The **start task** statement includes a **transaction** clause that allows multiple tasks to participate in a single transaction. When more than one task is participating in a single transaction, all tasks share the transactional locks. This is consistent with transactional locks being held by the transaction (as opposed to mutex locks, which are held by a task).

*Any* task in a transaction that holds even an exclusive write lock on an object is allowed to access and modify that shared transactional object. The tasks will not block each other at the level of transactional locks. However, the tasks will not concurrently be executing a method on the object or modifying an attribute; those tasks are regulated by the mutex lock.

## Transactional Objects Not in Transaction

Transactional objects are not required to be manipulated within the context of a transaction.

A task that is not executing in a transaction does not follow any of the rules for transactions. The task may modify a transactional object without incurring the overhead of logging and, in the case of a shared transactional object, the overhead of transactional locking.

Note that mutex locking rules always apply to shared objects, whether or not a task is executing in a transaction.

When designing transactions into your application, you might consider whether you can define tasks to concurrently execute outside of a transaction as well as tasks that should manipulate objects only under the control of a transaction.

## Cloning Transactional Objects

When you clone a transactional object, the resulting clone is not transactional. The `IsTransactional` attribute is not set, even if the `Transactional` class property default is set to on. Cloning a transactional object is useful to save the object's state at a particular time. If you wish to make the clone transactional, you must set the `IsTransactional` attribute after the clone as in the following example.

**Code Example 9-18** Setting the `IsTransactional` attribute for a cloned transactional object

```
self.OldName = self.Name.Clone(TRUE);  
self.OldName.IsTransactional = TRUE;
```

## Distributed Transactions and the Transactional Property

In iPlanet UDS Releases 2 and 3, the `Transactional` property is also used to determine if a distributed method is invoked in the context of a transaction or not. As a rule, you should allow the transactional property if you want a transaction to propagate to the partition that owns the object on which you're invoking a method. The iPlanet UDS transaction manager uses this property to determine if a transaction should propagate when a method is invoked on a distributed reference that is not a service object.

For example, a client begins a transaction and invokes a method on a service object. The method returns a reference to another, non-service object anchored in the remote partition. With respect to the client the returned object is a simple distributed reference, not a service object. Now, while still in the transaction, the client invokes a method on the returned object. This distributed method is only considered part of the transaction if the class definition of the returned object allowed the transactional property.

If the class definition of the returned object does not allow the transactional property, then methods invoked on distributed references to that object are not considered part of the transaction.

As a rule, methods invoked on distributed references to DB sessions and DB resource managers will always be part of the transaction as those classes allow the transactional setting.

This rule must be addressed when you construct a router and use customized logic to determine the object to use for the current request. In this case candidate objects are typically not service objects, and you should set their transactional property to achieve the desired behavior.

## Monitored Objects

A *monitored object* is an object that is mapped to a window widget. When changes are made to the value of such an object, the window display is automatically refreshed to display the changes. By default, the class Monitored property is allowed, indicating to iPlanet UDS that any object may be displayed.

If you know that a certain class will never be displayed, you should disallow the Monitored property. Turning off this property will increase the runtime efficiency of the class.

The Monitored property has no associated Object attribute, as the system internally decides whether or not to monitor an iPlanet UDS object.

Note that if Monitored is turned off for a class, you can still refresh the display after changing an object of that class by using the UpdateFieldFromData method of the Widget class (see the Display Library online Help for information about the Widget class).

## Monitored Objects

# Using Interfaces

This chapter provides complete information about how to use iPlanet UDS interfaces. The chapter provides a conceptual overview and then explains how you do the following:

- use interfaces with dynamic class loading
- use interfaces for multiple interface inheritance

## About Interfaces

An interface defines a set of class elements, without providing the code that implements them. The interface provides the method and event handler signatures that define a standard “interface” to an object. The code for the methods and event handlers in the interface is provided by the classes that *implement* the interface.

For example, the AdaptableAuction sample application defines a TaxCalculationIFace interface, which provides a method signature and an event related to calculating taxes on a sale. The method is called CalculateTax and the event is called TaxCalculated. The code for CalculateTax is provided by the TaxCalculationImp class, which implements the TaxCalculationIFace interface.

Using an interface separates the definition of a class from its implementations, allowing you to design your application using “component software” techniques. In iPlanet UDS, interfaces allow you to use both dynamic class loading and multiple interface inheritance. iPlanet UDS’s dynamic class loading feature enables you to deploy an application that uses an interface as a declared data type, and have developers in the various deployment environments plug in the appropriate class implementations for their environments. Multiple interface inheritance allows you to add generic operations, such as sorting, to your classes without restructuring your class hierarchy.

## Interface Elements

An interface has the same elements as a class, except for attributes. These elements include: virtual attributes, methods (method signatures), events, event handlers (event handler signatures), and constants. A *method signature* is the method name, parameter list, and return value. Likewise, an *event handler signature* is the event handler name and parameter list. See [“Interface Elements” on page 368](#) for complete information on these elements.

## Implementing an Interface

Implementing an interface in a class means providing the code for all methods and event handlers defined in the interface. Any number of classes can implement a single interface, which provides multiple implementations for a single interface. In addition, a single class can implement multiple interfaces. See [“Implementing an Interface” on page 364](#) for complete information on implementing an interface.

## Using an Interface as a Data Type

You can use an interface as the declared data type for any data item. The AdaptableAuction example uses the TaxCalculationIFace interface as the declared type of a local variable.

However, when you create the actual object associated with the data item, the object’s runtime type must be one of the classes that implement the interface. In other words, the implementing class is the data item’s runtime type. The AdaptableAuction example has a method that finds the implementing class for the interface, and creates an instance of that class. The runtime type of the object is the implementing class, either TaxCalculationImp or NewTaxCalculationImp, depending on which of two files is copied into dynload.dat when you run. (See [Appendix A, “iPlanet UDS Example Applications”](#) for complete information about the AdaptableAuction example.)

The classes that implement an interface can be included within your application code or you can load them at runtime from a shared library.



## Dynamic Class Loading

Loading the implementing classes at runtime enables you to customize the application for each deployment environment. Dynamic class loading enables you to create objects in your code from implementing classes that will be added to the environment *after* the original application is deployed. When the application is running, it dynamically loads the implementing classes provided in a shared library of the current environment. The application can then create objects from the dynamically loaded classes.

For example, the AdaptableAuction sample application reads data from a flat file to determine which implementation to load. At runtime, you can change the data in this file, and the application will load a different implementation. The data in the file tells the application which library and class to load.

For complete information on dynamic class loading, see [“Dynamic Class Loading” on page 371](#).

## Multiple Interface Inheritance

When a class implements an interface, it “inherits” its class elements from the interface, as well as from its own superclass, providing multiple interface inheritance. iPlanet UDS’s multiple interface inheritance feature is useful for adding generic functionality to existing classes, without restructuring your class hierarchy. The example described under [“Using Multiple Interface Inheritance” on page 389](#) shows how you could use an interface to add a generic Sort method to an existing class hierarchy. For complete information on multiple interface inheritance, see [“Using Multiple Interface Inheritance” on page 389](#).

## Polymorphism

Using interfaces for both dynamic class loading and multiple interface inheritance provides the benefit of polymorphism. Polymorphism means the ability of a data item to refer at runtime to any number of different objects. In TOOL, we make the distinction between the declared type and the runtime type. As described above, when you declare an item of an interface type, the interface that defines the data item is its declared type. However, when you assign an object value to the data item, the runtime type of the object assigned to the data item can be any class that implements the interface. In other words, the data item can take any number of forms (the term “polymorphism” is literally defined as the ability to assume many different forms).

The benefit here is that you can perform an operation on a data item whose type is an interface without knowing anything about how the object’s class has implemented the operation. As long as the class implements the interface, you are assured that the operation will work correctly for the object.

For example, the `AdaptableAuction` application uses the `TaxCalculationIFace` interface as the declared type of a local variable. The class that defines the object value for the local variable is loaded at runtime so the application does not know the runtime type for the local variable. However, because the interface defines the `CalculateTax` method and `TaxCalculated` event, the application can use `CalculateTax` and `TaxCalculated` to operate on the object.

## Interface Hierarchies

Unlike a class, which always has a superclass, an interface does not need a super-interface. When you create an interface, you choose whether or not the interface has a super-interface. If the interface does have a super-interface, it inherits all the interface elements defined for its super-interface, just as a class inherits from its superclass. If the interface does not have a super-interface, you must define the entire interface from scratch—it does not inherit any interface elements.

Setting up an interface hierarchy is similar to setting up a class hierarchy. Each sub-interface inherits all the interface elements defined for its super-interface. The sub-interface can overload methods by defining different parameter types for the same method name. However, overriding is not allowed. Because methods and event handlers in interfaces have no code associated with them, there is no way to override them. See [“Interface Elements” on page 368](#) for information about defining methods and event handlers in interfaces.

The following sections provide background information about creating an interface, providing implementations of the interface, and using the interface as a declared data type. [“Interface Elements” on page 368](#) provides detailed information about the individual elements in an interface.

## Creating an Interface

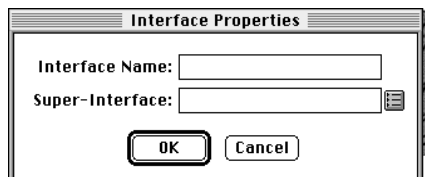
You define interfaces using the Interface Workshop. Creating an interface is similar to creating a class. You specify a name for the interface and, optionally, a super-interface. You then use the Interface Workshop to define each of the interface elements.

To create a new interface, you must start from the Project Workshop. The New Interface tool or the Component > New > Interface command creates a new interface with the name you specify.

► **To create an interface**

1. In the Project Workshop, click the New Interface tool or choose the Component > New > Interface command.

The Interface Properties dialog opens.



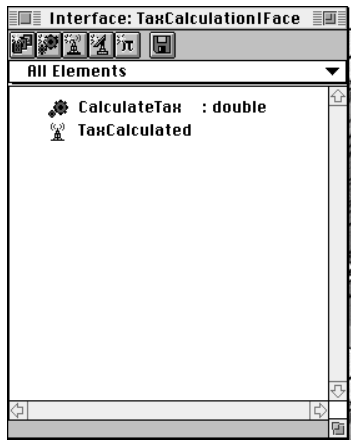
2. In the Interface Properties dialog, specify the name and, if desired, the super-interface for the interface. Click the OK button to create the interface.

Fill in the fields on the dialog as follows:

Interface Property	How to Specify It
Interface Name	Type in the interface name.
Super-Interface	Type in the super-interface name or use the browser button to display a list of interfaces from which you can make a selection.

After the Interface Workshop opens, you can define the individual elements in the interface. Defining the individual elements in an interface is the same as defining the individual elements in a class, except that you do not write code for the methods and event handlers. For complete information on using the Interface Workshop, see *A Guide to the iPlanet UDS Workshops*.

The following figure illustrates the complete definition of the TaxCalculationIFace interface from the iPlanet UDS AdaptableAuction example.

**Figure 10-1** TaxCalculationIFace Interface

## Implementing an Interface

To implement an interface, you declare one or more classes as “implementing” the interface. Then, in each class that implements the interface, you write the “implementation” code for all the methods and event handlers in the interface.

The implementing class must also define every event in the interface. You cannot exclude any of the methods, event handlers, or events, or you will get a compile error. Constants and virtual attributes do not need to be implemented in the implementing class.

When a class implements an interface, all of its subclasses also implement the interface. When you assign an object to the data item whose declared type is an interface, the object’s type can be a class that implements the interface or a *subclass* of a class that implements the interface.

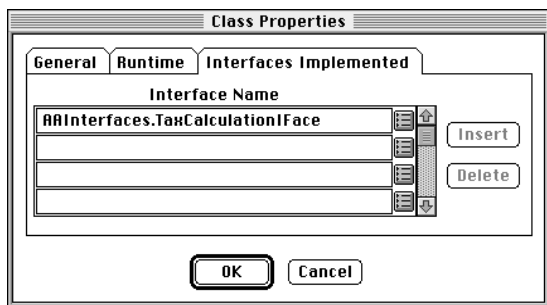
Remember, a single class can implement multiple interfaces. In addition, the class may also define functionality of its own. The multiple interface inheritance example described under [“Using Multiple Interface Inheritance” on page 389](#) shows classes that implement the Sortable interface to complement their basic functionality.

To implement an interface in a class, you use the Class Workshop. The Class Properties dialog for an individual class allows you to list one or more interfaces that the given class will be implementing. Any number of interfaces can be implemented by a given class, and any number of classes can implement a given interface.

The classes that implement the interface can be included within your application code or you can load them at runtime. See [“Dynamic Class Loading” on page 371](#) for information about how to load the implementing classes at runtime.

► **To implement an interface in the Class Workshop**

1. In the Class Workshop, choose the File > Properties... command.
2. On the Class Properties dialog, select the Interfaces Implemented tab.
3. On the Interfaces Implemented tab page, enter the interface name.



4. In the Project Workshop, open the interface for which you are providing the implementation.
5. In the Interface Workshop, drag each of the methods, event handlers, and events from the interface to the Class Workshop, and drop each one on the implementing class.

Copying the methods, event handlers, and events to the class conveniently defines all the elements that you need to implement in the class.

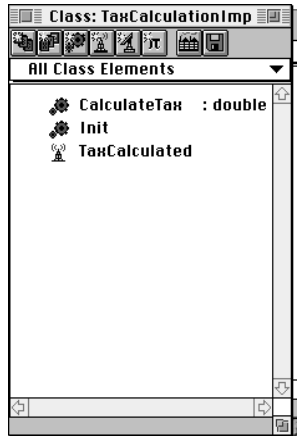
You can also copy any constants and virtual attributes you wish to include in the class; however, the constants and virtual attributes do not need to be implemented in the class.

6. Open each method, and write the implementation code using the Method Workshop.
7. Open each event handler, and write the implementation code using the Event Handler Workshop.

Every implementing class for an interface must provide implementation code for every method and event handler defined in the interface. The implementing class must also define every event in the interface. You cannot exclude any of the methods, event handlers, or events, or you will get a compile error. Constants and virtual attributes do not need to be implemented in the implementing class.

The following figure illustrates the full definition of the `TaxCalculationImp` class, which implements the `TaxCalculationIFace` interface in the iPlanet UDS AdaptableAuction example:

**Figure 10-2** TaxCalculationImp Implementation of TaxCalculationIFace Interface



## Implementing Multiple Interfaces

As described earlier, a single class can implement multiple interfaces. There is no limit on the number of interfaces that can be implemented by a single class. However, there are two restrictions of which you need to be aware.

The first restriction is that when a single class implements two interfaces that contain identical method or event handler signatures, the class can provide only a *single* implementation of that method or event handler. For example, if both the A and B interfaces contain an `Add` method, a class implementing both A and B can define only one `Add` method. In this case, you must be sure that the single implementation provides the correct functionality for both interfaces.

Of course, you must also watch out for this same problem if the superclass for a class and the interface it implements contain the same method signature.

If your class is implementing two interfaces that contain methods with the same names but with different parameter lists, you must implement both variations of the method in your class (that is, you must overload the method). If you do not, you will get an error, because your class is not correctly implementing both interfaces.

The second restriction is that your class cannot implement two interfaces that contain event handlers with the same names but with different parameter names. This restriction is necessary because you cannot overload event handlers; your class can contain only one event handler with a given name. Therefore, if your class specifies that it is implementing two interfaces that contain an event handler with the same name but with different parameter lists, you will get an error because your class is not correctly implementing both interfaces.

## Using an Interface as a Declared Type

With one exception, you can use an interface as a declared data type anywhere that you can use a class. The following example illustrates declaring a local variable with the `TaxCalculationIFace` interface as the data type:

**Code Example 10-1** Declaring a local variable using an interface as a data type

```
-- This method gets an instance of the needed class type, then
-- uses this instance to call methods implemented in the
-- implementation class.

newInstance : Object;
RI : RegInfo = new;

-- Get an instance of the implementation of the interface.
-- CreateImpInstance finds the library and class, and calls
-- InstanceAlloc(). It returns the value returned by
-- InstanceAlloc().
newInstance = RI.CreateImpInstance();

-- Cast the object returned by InstanceAlloc() to the interface.
taxClass : TaxCalculationIFace =
(TaxCalculationIFace)(newInstance);

-- You can now call methods in the implementation.
for i in 1 to 3 do
    SaleService.ListOfSales[i].Taxes =
        taxClass.CalculateTax(SaleService.ListOfSales[i]);
    ...
end for;
```

**Project:** AdaptableAuctionServices • **Class:** SaleMgr • **Method:** GetAllTaxes

The only time you cannot use an interface as a data type is when you are defining a service object. The service object's type must be a class, not an interface. However, an interface can be used as the type for one of the service object's attributes.

When you assign an object to the data item, the object's type must be a class that implements the interface (or a subclass of a class that implements the interface). The implementing class is the data item's runtime type. The example shown above uses the `CreateImpInstance` method to create the object using a dynamically loaded class. It then assigns this object to the `TaxClass` variable.

Because the interface is the declared type of the data item, the only operations available for the object are those defined by the interface. However, when you use one of the interface's method or event handlers on the object, iPlanet UDS uses the implementation provided by the object's class.

To access operations that are defined for the class but are not defined for the interface, you must first cast the object returned by the `InstanceAlloc` method to its runtime type. Casting an object from an interface to a class is exactly the same as casting an object from a superclass to a subclass. See the *TOOL Reference Manual* for complete information on casting.

## Interface Elements

The following sections provide conceptual information about the elements common to all interfaces:

- virtual attributes
- methods
- events
- event handlers
- constants

### Virtual Attributes

As described in *A Guide to the iPlanet UDS Workshops*, a virtual attribute does not store a value or point to an object—instead, a virtual attribute consists of two expressions, one that is evaluated when the program sets the value of the attribute and another that is evaluated when the program gets the value of the attribute. The Get expression for the virtual attribute is required, but the Set expression is optional. A virtual attribute without a Set expression is a read-only attribute.

Defining a virtual attribute in an interface allows you to make a method invocation look like an attribute. Typically, a virtual attribute provides a convenient way for getting and setting a complex value.





All methods used in the Get and Set expressions for the virtual attribute must be defined in the interface.

You do not need to “implement” the virtual attribute in the classes that implement the interface. When a data item’s declared type is an interface, iPlanet UDS always uses the virtual attribute definition provided by the interface. However, you can “re-define” the virtual attribute in the implementing classes. In this case, when the *declared type* of the data item is an interface, iPlanet UDS uses the definition provided by the interface, and when the *declared type* of the data item is the implementing class, iPlanet UDS uses the definition provided by the class.

## Methods



A method is a procedure that is specially written to operate on an object. Every method consists of a method signature (which specifies the method name, parameters, return type) and a statement block that contains the TOOL code that performs the operations on the object. In an interface, you define only the method signature; a method defined in an interface does *not* include source code. The class (or classes) that implements the interface provides the source code for the method.

The AdaptableAuction example provides two implementations of the TaxCalculationIFace interface. The interface defines the following method signature:

```
CalculateTax(theSale:Sale) :double
```

The two implementations of this method signature are in separate projects, in classes with different names. One implementation of CalculateTax takes into consideration whether the buyer was a non-profit organization; the other doesn’t.

For examples of the use of methods within interfaces, see the following methods in the AdaptableAuction example:

- CalculateTax (TaxCalculationImp class in AAImplementations project)
- CalculateTax (NewTaxCalculationImp class in AAImp2) project

You can overload a method in an interface. To overload a method, you simply add a new method signature using the same name with a different parameter list. The class (or classes) that implement the interface must provide source code for every method signature in the interface.

Because the interface defines only a method signature, overriding an inherited method has no effect. The inherited method and the overriding method have exactly the same signature.



## Events

An event is a signal that something has changed. Every event has a name and, optionally, one or more parameters. An event defined for an interface is exactly the same as an event defined for a class. You can use the **post** statement to post the event on any object whose class implements the interface.

All classes that implement the interface must implement all events defined in the interface. Implementing an event in a class consists of re-defining the event name and parameter list.

The `TaxCalculationIFace` interface in the `AdaptableAuction` example defines an event called `TaxCalculated`. Both implementations of this interface define the `TaxCalculated` event.

For further information on using events in TOOL, see the *TOOL Reference Manual*.



## Event Handlers

An event handler is a named block of TOOL code that provides programming to be executed in response to one or more events. The event handler provides reusable, modular event handling code that you can include in any number of **event** statements.

In a class, an event handler consists of an event handler signature, which consists of the event handler name and parameters, and the event handler source code. In an interface, you define only the event handler signature; an event handler defined in any interface does *not* include source code. The class (or classes) that implements the interface provides the source code for the event handler.

Unlike methods, there is no overloading for event handlers. There can only be one event handler with a given name in the interface.

Because the interface defines only an event handler signature, overriding an inherited event handler has no effect. The inherited event handler and the overriding event handler have the exact same signature.

*A Guide to the iPlanet UDS Workshops* contains detailed information about event handlers. For information on using event handlers in TOOL, see the *TOOL Reference Manual*.



## Constants

A constant is a literal string or numeric value that has a name. When you declare the named constant, you specify a constant name and a value. You can then use the constant name in place of the value in the TOOL code that implements the interface's methods and event handlers.

The most common use for a constant within an interface is for specifying the values of a parameter.

When a constant is defined within an interface, the implementing code within classes that implement the interface can reference the constant directly. Other classes must reference the constant with the following syntax:

*interface\_name.constant\_name.*

Remember, although you can use constants to specify values in your TOOL code, you cannot use them to specify values in dialogs in the iPlanet UDS Workshops.

You do not need to “implement” the constant in the classes that implement the interface. When a data item’s declared type is an interface, iPlanet UDS always uses the constant definition provided by the interface. However, you can “re-define” the constant in the implementing classes. In this case, when the *declared type* of the data item is an interface, iPlanet UDS uses the definition provided by the interface, and when the *declared type* of the data item is the implementing class, iPlanet UDS uses the definition provided by the class.

## Dynamic Class Loading

iPlanet UDS’s dynamic class loading feature allows you to dynamically access classes in a library that has been deployed in the current environment. With dynamic class loading, you can use an interface as a declared type in your code and then, at runtime, dynamically load the class that implements the interface in order to create the actual object itself. You load the implementing class from a shared *implementation library* that must be deployed in the same environment as the application.

Using dynamic class loading for interfaces is a two-phase effort:

1. The application that uses the interfaces must be developed and deployed.
2. At one or more deployment sites, the interfaces must be implemented, and the classes that implement the interfaces must be deployed in an implementation library.

At each deployment site, the combination of the *deployed application* (as the original developer created it) and the *deployed implementation library* (which contains the “customized” class implementations) work together to create the functionally complete application.

Typically, using dynamic class loading requires cooperation between two different developers:

- the application developer

The application developer defines the interface, uses the interface within the application code, and writes code to dynamically load the implementing classes at runtime.

- the class implementer

The class implementer defines the classes that implement the interface. The library that contains these implementing classes must be deployed in the same environment where the application is deployed.

The following sections provide step-by-step instructions for both these developers.

## Application Developer: Using Dynamic Loading within Application Code

When you are writing an application that uses dynamic class loading, you not only need to provide the code that dynamically loads the implementing classes for the interface, but you must also create an *interface library* and set up a mechanism that will enable the class implementer to register the implementation libraries that he creates.

### ► To use dynamic class loading with an interface

1. In a separate project from the rest of your application, define the interface and any classes needed by the interface. Make this project a supplier to the project that defines your application.

2. Set up a flat file or database outside the iPlanet UDS application where implementers can register the class libraries that implement the interface.

The developers who implement the class must register the implementation libraries they create using the mechanism that you provide for them.

3. In your application, write the code that loads the library, finds the class definition, and assigns the object created from the class to a data item with an interface type.

To load the library, use the FindLibrary method on the Partition object. To find the class definition in the library, use the FindClass method on the Library object. To create the object from the class, use the InstanceAlloc method on the ClassType object.

4. Make a library from the project that defines the interface.

The developers who are going to implement the interface need to use the interface library in order to share the interface definition.

5. Test the application.

To test an application that uses dynamic loading, you must provide a default implementation for the interface.

6. Deliver the following to the class implementers:

- a. the application distribution
- b. the interface library
- c. instructions on how to register the implementation library that the class implementers will create to provide the appropriate implementations
- d. a test driver for the interface

The following sections provide detailed information about each of the above steps.

## Step 1. Defining the Interface

You must begin by creating a separate project to contain the interface. The interface must be in a separate project because when you are ready to deploy your application, you must make the interface project into a library (the *interface library*). You will then deliver this interface library to all the developers who will implement the interface.

The project that defines the interface may also need to define *helper classes* for the interface. Helper classes are classes that you use for the parameter types in the method and event handlers defined in the interface. The interface library that you deliver to the developers who are implementing the interface must also contain any classes that they need to use within the implementation code.

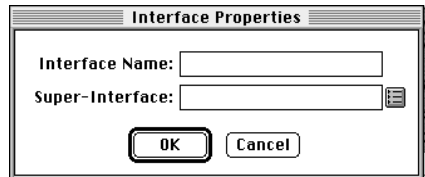
After creating the new project to contain the interface, you must make the interface project a *supplier* to the project that defines your application. In the AdaptableAuction example, the AAInterfaces project defines the TaxCalculationIFace interface. The AAInterfaces project is a supplier to the AdaptableAuctionServices project, which defines the AdaptableAuction application.

Finally, you can define the interface itself using the Interface Workshop. In the Project Workshop, the New Interface tool or the Component > New > Interface command creates a new interface with the name you specify.

► **To create an interface**

1. In the Project Workshop, click the New Interface tool or choose the Component > New > Interface command.

The Interface Properties dialog opens.

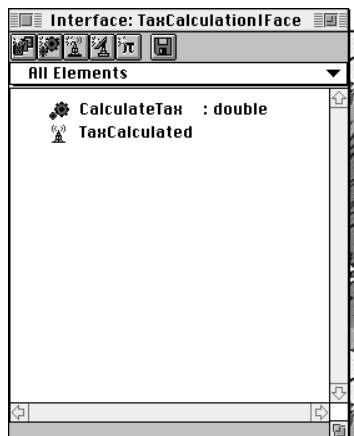


2. In the Interface Properties dialog, specify the name and, if desired, the super-interface for the interface. Click the OK button to create the interface.

After the Interface Workshop opens, you can define the individual elements in the interface. Creating an interface in the Interface Workshop is the same as creating a class in the Class Workshop, except that you do not provide source code for the methods and event handlers. See *A Guide to the iPlanet UDS Workshops* for detailed information about creating an interface.

The following figure shows the definition of the TaxCalculationIFace interface in the iPlanet UDS AdaptableAuction example.

**Figure 10-3** Interface Definition



## Step 2. Providing the Mechanism for Registering Implementing Classes

The programmers who are implementing the interface must be able to notify your application about the library that contains the class that implements the interface. Your application needs to obtain the following information:

- project name for project that defined the implementation library
- distribution ID for library distribution (derived from the project name)
- compatibility level for the library distribution
- library name of the library
- class name

In turn, your application must be able to check the current environment for this same information before it can load the library that contains the implementing class. Therefore, you should set up a mechanism outside of your application, such as a database or flat file, which developers can use to register their implementation libraries and which your code can check for this information.

For example, the AdaptableAuction example reads data from a flat file called `dynload.dat`. Two data files are provided with the example: `dynload1.dat` and `dynload2.dat`.

This is the data stored in `dynload1.dat`:

```
AAImplementations //Project name
aaimplem //Distribution ID
0 //Compatibility level
aaimplem //Library name
TaxCalculationImp //Class name
```

This is the data stored in `dynload2.dat`:

```
AAImp2 //Project name
aaimp2 //Distribution ID
0 //Compatibility level
aaimp2 //Library name
NewTaxCalculationImp //Class name
```

When you run the AdaptableAuction example, you can copy dynload1.dat to dynload.dat. The data in dynload.dat is read in, and this implementation will be used to calculate the taxes. To try out the example's second implementation, you can copy dynload2.dat to dynload.dat while the application is running. The next time a tax calculation is performed, the second implementation will be used.

Optionally, you may want to read data from a database, or use a combination of reading data from a database and reading from a flat file. The flat file is used in the example for simplicity.

The particular mechanism you use is up to you. However, when you deploy your application, you must be sure to give careful instructions to the programmers implementing the interface about how to register their class libraries.

## Step 3. Loading the Class and Creating the Object

After you have created the interface, you can use it within your code as the declared data type for any appropriate data item. However, when you are ready to create the actual object that is associated with the data item, you must provide the code that will load the implementing class.

To load the implementing class, you must use the Library class, which allows you to dynamically access classes in a library that has been deployed in the current environment. The FindLibrary method in the Partition class allows you to get access to the Library object that contains the implementing class. The FindClass method in the Library class allows you to get access to the ClassType object associated with the implementing class.

The following procedure describes the basic steps you must follow in your code to load the class and create the object. These steps are followed by detailed information about the FindLibrary and FindClass methods. The end of this section contains sample code that illustrates use of these methods.

### ► To load the class and create the object

1. Get the library registration information from the flat file or database where the implementer is going to register the implementation library.
2. Use the FindLibrary method on the Partition object to get access to the Library object that contains the implementing class.
3. Invoke the FindClass method on the Library object to return the ClassType object associated with the class.



4. Use the InstanceAlloc method to instantiate the object using the ClassType object.

The InstanceAlloc method allows you to create an object when you do not know what the class will be until runtime. The result of InstanceAlloc is a generic Object, so you typically need to cast the result to the interface type.

The FindLibrary method in the Partition class locates the specified library, loading it on the partition if necessary, so that you can access the classes stored in the library. The syntax is:

---

**FindLibrary** (*projectName=TextData, distributionID=TextData, compatibilityLevel=integer, libraryName=TextData, loadOptions=uj4*)

---

**Returns** Library

Parameter	Required	Input	Output
projectName	●	●	
distributionID	●	●	
compatibilityLevel	●	●	
libraryName	●	●	
loadOptions	●	●	

---

The FindLibrary returns a Library object, which provides you with access to the library from your code. If the library is already loaded in the current partition, the FindLibrary method simply returns the Library object to provide you access to the library. If the library is not loaded in the partition, the FindLibrary method automatically loads the library into the partition.

Typically, the values for the projectName, distributionID, compatibilityLevel, and libraryName parameters come from the flat file or database outside the iPlanet UDS application, where developers who have implemented the interface will register the libraries that they will be deploying.

The loadOptions parameter specifies loading options for the library. By default, if the library is not already loaded in the partition, iPlanet UDS automatically loads the library. iPlanet UDS automatically unloads the library when the partition no longer need to access it.

See the Framework Library online Help for complete information on the FindLibrary method.

Once you have returned the appropriate Library object, you can invoke the FindClass method on it to return the ClassType object associated with the class. The FindClass method returns the ClassType object associated with the specified class within the current library. The syntax is:

---

**FindClass** (className=*TextData*)

---

Returns ClassType

Parameter	Required	Input	Output
className	●	●	

---

The FindClass method returns a ClassType object, which you can use with the InstanceAlloc method to create an instance of the class. If the class does not exist, the FindClass method returns NIL.

Note that after using the InstanceAlloc method to create an instance of the class, you will need to cast it. The InstanceAlloc method returns a generic Object. If you cast the object to an interface, iPlanet UDS checks that the object's class actually implements the interface. (See the Framework Library online Help for complete information on the InstanceAlloc method.)

See the Framework Library online Help for complete information on the FindClass method.

The following sample code illustrates use of the FindLibrary and FindClass methods:

**Code Example 10-2** Using the FindLibrary and FindClass methods

```
-- Load the implementation and create an instance from the
-- RegType. Read Library and Class information from a file.

className : TextData = new;

-- Check whether implementation library is already loaded.
if CurLib = nil then
  -- Read in library information from file or database.
  file_to_read : File = new;
  file_to_read.SetPortableName(
    (name = %{FORTE_ROOT}/install/examples/frame/dynload.dat');
  file_to_read.Open(aAccessMode = SP_AM_READ);

  LibDesc.ProjectName = new;
  file_to_read.ReadLine(
```

**Code Example 10-2** Using the FindLibrary and FindClass methods (*Continued*)

```

        target = LibDesc.ProjectName,
        includeEOL = FALSE);
LibDesc.ApplicationID = new;
file_to_read.ReadLine(
    target = LibDesc.ApplicationID,
    includeEOL = FALSE);
td : TextData = new;
file_to_read.ReadLine(
    target = td,
    includeEOL = FALSE);
LibDesc.CompatibilityLevel = td.IntegerValue;
LibDesc.LibraryName = new;
file_to_read.ReadLine(
    target = LibDesc.LibraryName,
    includeEOL = FALSE);
file_to_read.ReadLine(
    target = className,
    includeEOL = FALSE);
file_to_read.Close();

-- Find the library.
CurLib = task.Part.FindLibrary(LibDesc.projectName,
    LibDesc.applicationID,
    LibDesc.compatibilityLevel,
    LibDesc.libraryName);
-- In some cases this exception will be handled by Forte
-- before you get here. It's useful to put this exception
-- handling in, for the cases where Forte won't catch it.
if CurLib = nil then
    ex : SystemResourceException = new(
        ReasonCode = SP_ER_LIBLOADFAIL,
        Severity = SP_ER_USER,
        Message = 'Unable to load the library. ');
    task.ErrorMgr.AddError(ex);
    raise ex;

    end if;
end if;

-- Load the class type.
if RegType = nil then
    RegType = CurLib.FindClass(className = className);
    if RegType = nil then
        ex : SystemResourceException = new(
            ReasonCode = SP_ER_LIBLOADFAIL,
            Severity = SP_ER_USER,
            Message = 'Unable to find the class. ');
        task.ErrorMgr.AddError(ex);
        raise ex;

        end if;
    end if;
end if;

```

**Code Example 10-2** Using the FindLibrary and FindClass methods (*Continued*)

```
return RegType.InstanceAlloc ();
```

**Project:** AdaptableAuctionServices • **Class:** RegInfo • **Method:** CreateImpInstance

## Step 4. Making the Interface Library

After your interface is defined, you must create the interface library needed by the class implementers.

To create the library, open the project that contains the interface, and follow the standard procedure for creating a library. In the Partition Workshop, create a library distribution for each environment in which the application will be deployed. This process for creating a library is described in *A Guide to the iPlanet UDS Workshops*.

The only special issue you need to consider for the interface library is that if the partition where the code that uses the interface is running is a compiled partition, the interface library must also be compiled.

### ► To configure a library

1. Open the project you wish to configure as a library.
2. In the Project Workshop, choose the File > Configure as > Library command.  
The Partition Workshop opens.
3. If necessary, select the deployment environment from the environment drop list.

Partitioning an interface library and making the interface library distribution is the same as creating an ordinary library distribution. See [Chapter 8, “Deploying iPlanet UDS Applications and Libraries”](#) for information about creating a library distribution.

If your interface library uses supplier projects containing helper classes, these supplier projects must also be configured as libraries and given to the class implementers. The easiest technique is to add all these supplier libraries to the same library distribution that contains the interface library.

## Step 5. Testing the Application

To test an application that uses dynamic loading, you must provide a default implementation for the interface. Follow the instructions under “**Class Implementer: Providing Implementations for Dynamic Loading**” on page 382 and deploy the implementation library within your development environment.

Your application will not run correctly unless you have deployed the implementation library in your development environment and have registered the library information so your application can access the library.

## Step 6. Delivering the Application and Interface Library

To provide an implementation for an interface in your application, the developers who are creating classes that implement the interface need the following items from you:

- the application distribution  
The implementer needs the application distribution so he can test his implementations by running the complete application.
- the interface library distribution  
The implementer must install the interface library distribution into his own development environment and import the interface library into his repository. He must then include the library as a supplier to his own project so he can create a class that implements the interface.
- instructions for registering the implementation library with your application  
After the class implementer creates his implementation library and it is deployed in the environment where your application is running, the system manager must register the implementation library so that your application can access and load the library.
- a test driver for the interface  
If possible, it is a good idea for you to write a separate test driver for the class implementation, which the implementer can use to check his work. The test driver would invoke the interface methods and verify that the results are correct.

## Class Implementer: Providing Implementations for Dynamic Loading

Before you can provide implementations for the interface, you need two items from the application developer: the application distribution that uses the interface and the interface library that the application developer created for you. You also need instructions from the application developer about how to register the class library that you will be creating.

### ► To provide implementations for the interface

1. Import the interface library into your development repository.
2. Create a project to contain the class that implements the interface. Make the interface library a supplier to this project.
3. Create a class that implements the interface.  
See [“Implementing an Interface” on page 364](#) for information on implementing an interface.
4. Configure the project that defines the implementing class as a library and make a distribution for it.
5. Deploy the library distribution in the same environment where the application that uses the dynamic loading is deployed.
6. According to the application developer’s instructions, register the information about the new library in the designated flat file or database.
7. Test the implementation.

The following sections provide detailed information about each of these steps.

## Step 1. Importing the Interface Library

Before you can create the implementation for an interface, you must have access to the interface itself. The application developer who created the interface must provide you with an interface library that contains the interface definition. As described under “[Step 1. Defining the Interface](#)” on page 373, the interface library may also contain “helper” classes that define the method and event handler parameter types used in the interface. You will need to use these helper classes when you are implementing the interface.

---

**CAUTION** If you have access to the project that defines the interface, do *not* import its .pex file into your repository. When you are providing implementations for an interface that is using dynamic class loading, you must use the interface *library* as your supplier plan and not the original *project*. Therefore, you must import the .pex file for the library, not for the original project. If your repository already contains the project, you must either delete the original project from your repository before importing the library, or you must use a different repository.

---

You must install the interface library distribution in your development environment on all the nodes where it is needed. Installing a library distribution is the same as installing an application. You can copy the distribution directory and its subdirectories or use a CD or tape for transfer to a remote site. For instructions on installing a library distribution, see the *iPlanet UDS System Management Guide*.

After the library distribution is installed in your development environment, you must import the individual interface library (or libraries) that it contains into your development repository. To import a library into your repository, use the Plan > Import command in the Repository Workshop.

The .pex file for the library contains the library definition; this is the file you must import.

### ► To import a library

1. In the Repository Workshop, choose the Plan > Import command. In the file selection dialog, specify the name of the .pex file that contains the interface library.
2. After the library has been imported into your workspace, use the Integrate Workspace command to add the new library to the system baseline.

After the interface library has been added to your workspace, you will see the library in the Repository Workshop's browser. A library icon indicates that it is a library.

If the interface library uses supplier libraries, you must be sure to import all these supplier libraries into your development repository. You will need *all* the libraries in order to implement the interface.

Note that you must ensure that the interface library provided by the application developer and the implementation library that you create are always consistent. If the interface library changes, you must re-import the interface library into your repository and then create a new implementation library. You must then deploy the new implementation library in the environment where the application is running.

## Step 2. Creating the Implementation Project

You must create a separate project to define the implementing class. After you define the implementing class within this project, you will make a library from the project so the library can be deployed within the environment where the application will run. The library that contains the implementing class is the library from which the application will be dynamically loading the class.

After creating the new project, you must include the interface library provided by the application developer as a supplier plan for your new project.

### ► To include a library as a supplier

1. Make sure that the library is included in your workspace.
2. In the Project Workshop, choose the File > Supplier Plans... command.
3. In the Supplier Plans dialog, move the library from the list of available plans to the list of suppliers.

## Step 3. Creating the Implementing Class

To implement an interface, you use the Class Workshop. The class that implements an interface can be an existing class or you can create a new class.



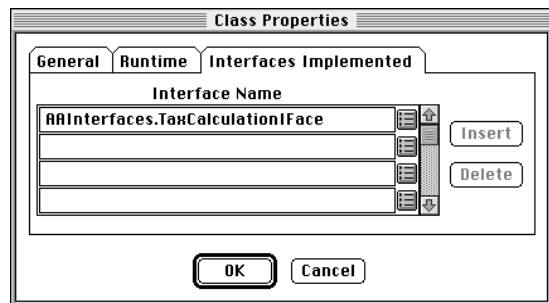
To indicate that a given class implements an interface, you enter the interface name into the class definition, as part of the class properties. Then, using the Class Workshop, you write the “implementation” code for all the methods and event handlers in the interface. You can implement a given interface in any number of classes.

There is only one restriction you should consider when writing the implementation code for the interface. If the interface is running on a server, you cannot open windows or dialogs. You can create objects from the Display library classes, however, you cannot display them on a server.

In the Class Workshop, the Class Properties dialog for an individual class allows you to list one or more interfaces which the given class will be implementing. Any number of interfaces can be implemented by a given class. The following instructions outline the steps required to implement one interface in one class.

► **To implement an interface in the Class Workshop**

1. In the Class Workshop, choose the File > Properties... command.
2. On the Class Properties dialog, select the Interfaces Implemented tab.
3. On the Interfaces Implemented tab page, enter the interface name.



4. In the Project Workshop, open the interface for which you are providing the implementation.
5. In the Interface Workshop, drag each of the methods, event handlers, and events from the interface to the Class Workshop, and drop each one on the implementing class.

Copying the methods, event handlers, and events to the class conveniently defines all the elements that you need to implement in the class.

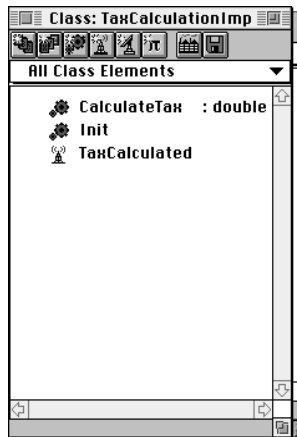
You can also copy any constants and virtual attributes you wish to include in the class; however, the constants and virtual attributes do not need to be implemented in the class.

6. Open each method, and write the implementation code using the Method Workshop.
7. Open each event handler, and write the implementation code using the Event Handler Workshop.

Every implementing class for an interface must provide implementation code for every method and event handler defined in the interface. The implementing class must also define every event in the interface. You cannot exclude any of the methods, event handlers, or events, or you will get a compile error. Constants and virtual attributes do not need to be implemented in the implementing class.

The following figure illustrates the full definition of the TaxCalculationImp class, which implements the TaxCalculationIFace interface.

**Figure 10-4** TaxCalculationImp Implementation of TaxCalculationIFace Interface



## Step 4. Making the Implementation Library

When you have finished defining the class that implements the interface, you are ready to make the project that contains the implementation into a library.

To create the implementation library, open the project that contains the class that implements the interface, and follow the standard procedure for creating a library. This process is described in *A Guide to the iPlanet UDS Workshops*.

The only special issue you need to consider for the implementation library is that you cannot make the implementation library compiled if the interface library was not compiled. If the interface library was interpreted, the implementation library must also be interpreted. If the interface library was compiled, the implementation library can be either interpreted or compiled.

► **To configure a library**

1. Open the project you wish to configure as a library.
2. In the Project Workshop, choose the File > Configure as > Library command.  
The Partition Workshop opens.
3. If necessary, select the deployment environment from the environment drop list.

Partitioning an implementation library and making the implementation library distribution is the same as creating an ordinary library distribution. See [Chapter 8, “Deploying iPlanet UDS Applications and Libraries”](#) for information on creating library distributions.

If the implementation library uses supplier projects, these supplier projects must also be configured as libraries and deployed in the environment where the application is going to run. The easiest technique is to add all these supplier libraries to the same library distribution that contains the implementation library.

## Step 5. Deploying the Implementation Library

After you make the library distribution, you must deploy it within the environment where the application is going to run. Installing a library distribution is the same as installing an application. You can copy the distribution directory and its subdirectories or use a CD or tape for transfer to a remote site. For instructions on installing a library distribution, see the *iPlanet UDS System Management Guide*.

Note that you must ensure that the interface library provided by the application developer and the implementation library that you create are always consistent. If the interface library changes, you must re-import the interface library into your repository and then create a new implementation library. You must then deploy the new implementation library in the environment where the application is running.

## Step 6. Registering the Implementation Library

The application developer will have provided you with instructions about how to register your implementation library in a flat file or database. The application's code checks the flat file or database to get the following information about the library that contains the implementing class:

- project name
- distribution ID (derived from the project name)
- compatibility level
- library name of the library
- class name

The particular mechanism used for registering the implementation library is up to the individual application developer. However, you must be sure to follow the developer's instructions to register your implementation library so the application code can use the library.

For example, the AdaptableAuction example reads data from a flat file called `dynload.dat`. Two data files are provided with the example: `dynload1.dat` and `dynload2.dat`.

This is the data stored in `dynload1.dat`:

```
AAImplementations //Project name
aaimplem //Distribution ID
0 //Compatibility level
aaimplem //Library name
TaxCalculationImp //Class name
```

This is the data stored in `dynload2.dat`:

```
AAImp2 //Project name
aaimp2 //Distribution ID
0 //Compatibility level
aaimp2 //Library name
NewTaxCalculationImp //Class name
```

When you run the AdaptableAuction example, you can copy dynload1.dat to dynload.dat. The data in dynload.dat is read in, and this implementation will be used to calculate the taxes. While the application is running, you can copy dynload2.dat to dynload.dat. The next time a tax calculation is performed, the second implementation will be used.

## Step 7. Testing the Interface

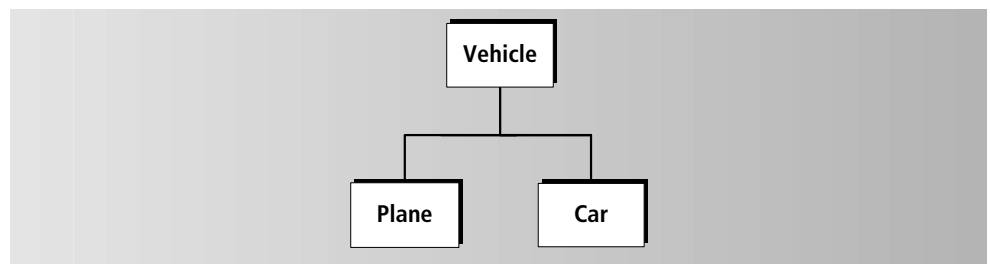
To test your implementation of the interface, you should first run the test driver provided by the application developer. This lets you detect regressions in the implementation.

When you have tested the implementation individually, you should test the interface with the application. To test the interface with the application, you must ensure that both the application and implementation library are installed in your development environment, and that you have registered the library information so the application can access the library. You can then simply run the application.

# Using Multiple Interface Inheritance

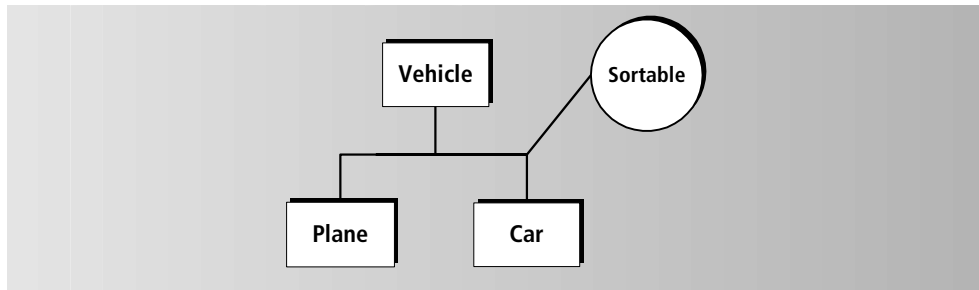
In iPlanet UDS, a given class can have only one superclass. The class inherits the attributes, virtual attributes, methods, events, event handlers, and constants defined by only one superclass. This is *single inheritance*. **Figure 10-5** illustrates single inheritance. The Car class inherits from a single superclass, Vehicle:

**Figure 10-5** Single Inheritance



When a class implements an interface, it can be said to “inherit” the virtual attributes, methods, events, event handlers, and constants defined by the interface. A class always inherits from its superclass. When a class implements an interface, it inherits class elements both from its superclass and from the interface that it implements. This provides a form of multiple inheritance called *multiple interface inheritance*. **Figure 10-6** illustrates multiple interface inheritance. The Car class inherits from its superclass Vehicle and also implements the Sortable interface:

**Figure 10-6** Multiple Interface Inheritance



Using interfaces for multiple inheritance is not the same as using multiple class inheritance. Specifically, using interfaces for multiple inheritance does not provide the benefit of re-usability. When a class implements an interface, it does not inherit any method or event handler source code, so it cannot “reuse” inherited elements. However, using interfaces for multiple inheritance provides the benefit of polymorphism as described under “**About Interfaces**” on page 359.

As Stephen McHenry says in his paper “Multiple Inheritance Using Interfaces”:

“With interface inheritance, the subclass is simply inheriting the ability to respond (in a polymorphic way) to exactly the same message as the superclass, or other subclasses. Thus, callers may utilize the methods of the subclass when addressing collections of the superclass in a polymorphic way.”

(We recommend reading this paper for background information on multiple interface inheritance. It is available on the Web at [www.softi.com/documents.html](http://www.softi.com/documents.html).)

## Declared Type and Runtime Type for Interfaces

When you declare an item of an interface type, the interface that defines the data item is its *declared type*. However, the class of the actual object associated with the data item, its *runtime type*, can be any class that implements the interface. This allows you to create a generic data item, such as `Sortable`, that represents a wide range of possible objects, for example, cars, fruit, and employees, all of which are sortable.

Using an interface as a data type means that you can perform an operation on the data item without knowing anything about how the object's class has implemented the operation. As long as the class implements the interface, you are assured that the operation will work correctly for the object.

## Example of Multiple Inheritance

For example, let's say you want to add the ability to sort an array to your current class hierarchy. You could create a `Sortable` interface that defines the following `Compare` method:

```
integer=Compare(target=Object)
```

The `Car`, `Fruit`, and `Employee` classes could each implement the `Sortable` interface, providing their own implementations for the `Compare` method. Of course, the code for the `Compare` method is different for each of these classes; `make`, `model`, and `year` are used to compare cars, `name` is used to compare fruits, and `identification number` is used to compare employees. However, invoking the `Compare` method is exactly the same.

You could then define a class called `ArraySorter`, which would sort any array of `Sortable` objects. It would define the following `Sort` method, which takes an array of `Sortable` objects as its input parameter, sorts the objects, and returns a new, sorted array:

```
Array of Sortable=Sort(source=array of Sortable)
```

The code for the `Sort` method would invoke the `Compare` method on the `Sortable` objects. Because each class implements the `Compare` method appropriately for itself, there is no need for the `Sort` method to have detailed knowledge about the objects it is sorting.

Since interfaces do not provide inheritance of source code, you might wonder why you wouldn't just add the new functionality to your existing class hierarchy through the use of superclasses, which do provide inheritance of source code.

In his paper “Multiple Inheritance Using Interfaces,” Stephen McHenry explains that even in systems that provide multiple class inheritance, multiple interface inheritance is a more effective technique because most operations that you would inherit from a second superclass are those that need to be implemented differently for each class that inherits them.

“Upon casual observation, it may seem like it is a nuisance to reimplement display for each of the classes that inherit the interface. It would be easier to just provide an implementation in the DisplayObject class and inherit that implementation. However, upon closer inspection, we find that each class that inherits the display operation must do something a little different depending upon the type of the class. So, while this means that we can treat collections of DisplayObject uniformly (by sending them all a display message), it makes no sense to inherit an implementation that must be overridden for each subclass. In this case, interface inheritance is perfectly adequate.”

In addition, using interfaces enables you to pick and choose the individual classes in the hierarchy that should implement the interface. Adding the same functionality to an existing class hierarchy might not work, without your having to make disruptive changes to the hierarchy. For example, not all objects are sortable, therefore, trying to add a generic Compare method to the Object class would not be effective. However, creating a Sortable interface allows you to select individually the classes that can be grouped into the “sortable” category; only the classes that implement the Sortable interface are “sortable.”

## Signature Conflicts

As described under [“Implementing an Interface” on page 364](#), a single class can implement multiple interfaces. Thus, a single class can “inherit” from one superclass and any number of interfaces. Remember, the potential problem here is that the superclass and the interfaces being implemented by a single class might contain identical method or event handler signatures. The class can provide only a *single* implementation of that method or event handler. Therefore, you must be sure that the single implementation provides the correct functionality for both interfaces.



# Working With Service Objects

This chapter provides conceptual information about service objects, including:

- setting service object visibility
- setting service object dialog duration
- providing failover
- providing load balancing
- using reference partitions to share service objects between applications
- setting the service object's environment search path

For information about defining and partitioning service objects and modifying their definitions within a particular configuration, see *A Guide to the iPlanet UDS Workshops*.

## About Service Objects

iPlanet UDS service objects provide the basis for creating an iPlanet UDS distributed application. All of the distributed services with which your application interacts are represented by service objects. For example, to access a database from your application, you create a service object to represent that database. Likewise, to access an external application from an iPlanet UDS application, you create a service object to represent the external application.

A *service object* is a named object that represents an existing external resource, an iPlanet UDS shared business service that is shared by multiple users, or a service that is replicated to provide failover or load balancing. The service object contains information needed by the service as well as operations that the service can perform.

When you create a service object, you name it so that it is accessible throughout the distributed application. Any other services in the application can interact with the service object in the same way that any two objects can interact, by invoking methods and posting events.

In the Project Workshop, you declare all the service objects to be included in the project. Later on, when you partition the application in the Partition Workshop, iPlanet UDS assigns the service objects in the main project and all its supplier projects to particular nodes in the environment.

There are three kinds of service objects. The class you specify for the service object determines which kind it is:

Service Object	Class	Description
TOOL Class	Any custom class (including C classes) or any applicable iPlanet UDS class	Represents a user-defined service.
DBResourceMgr	DBResourceMgr	Represents a DBMS installation.
DBSession	DBSession	Represents a database session for a particular database resource manager.

The following sections provide general information about these three kinds of service objects.

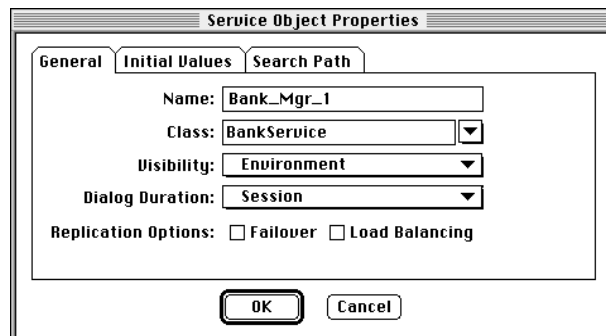
## TOOL Class Service Objects

A TOOL class service object is an iPlanet UDS shared service that you define based on either a custom class (a class that you have designed and defined on your own) or on a class provided in an iPlanet UDS library or supplier project. Examples of services that you could create from a TOOL class include a phone call tracking service, a document control service, and an auction manager, as in the Auction application. This chapter uses these example services in the following sections to illustrate service objects concepts.

Because a service object interacts with objects in remote partitions, the class from which you create a service object must be distributed (that is, the class must have its Distributed property set to Allowed). The iPlanet UDS class reference documentation indicates which classes are distributed and can therefore be used for service objects. If you are creating your own class to define a service, you must be sure to set the Distributed property for the class to Allowed or IsDefault (see [“Distributed Objects” on page 330](#) for information).

[Figure 11-1](#) shows the General Properties tab page for a TOOL service object. In fact, the General Properties tab page is the same for all service objects.

**Figure 11-1** TOOL Class Service Object General Properties Tab Page



When you create a TOOL Class service object, you can specify initial values for any of the public attributes in the object that have simple data types. Any public attributes for which you do not specify a value are set to their default values.

## Service Objects for Database Access

Two types of service objects enable an iPlanet UDS application to access a relational database. These two service objects have different advantages and restrictions; for a more detailed explanation of how they differ and how to use them, see the manual *Accessing Databases*.

The primary difference between the two service objects is that the DBSession service object definition includes a user name and password to be used for every database connection; thus all connections made through this type of service object are made for the “same” database user (and to the same actual database). In

contrast, the DBResourceMgr service object supports database connections from any valid user name/password combination, and can be used to connect to multiple databases (of the same vendor). Note that both types of service objects, however, do support multiple connections.

Before you can use either of these service objects, your system manager must use the Environment Console to define a *resource manager name* for the database(s) you wish to access. The resource name represents one particular database vendor type (for example, Oracle, Sybase, Informix, DB2, ODBC, Ingres) that is available for the designated node. Both types of service objects communicate with a particular database through a resource manager appropriate to the database vendor.

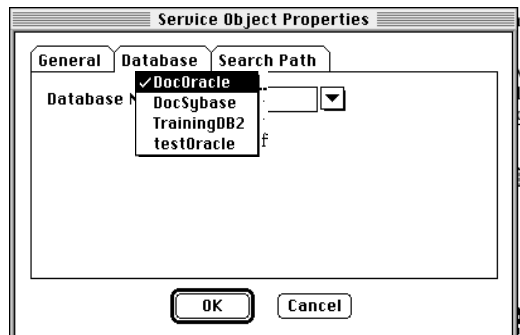
For either service object you can override the resource name for a particular configuration in the Partition Workshop.

## DBResourceMgr Service Objects

A DBResourceMgr service object represents one type (vendor type) of database. After the DBResourceMgr service object is created, you can start multiple connections (sessions) to the database(s) that it represents.

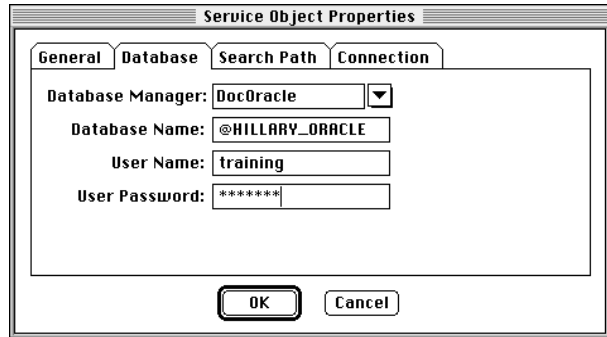
As shown in [Figure 11-2](#), when you create a DBResourceMgr service object, you specify the name of the resource manager that the service object will use. As shown, iPlanet UDS provides a drop list of resource manager names from which you can choose, or you can enter a name that will be defined later.

**Figure 11-2** DBResourceMgr Service Object Database Tab Page



## DBSession Service Objects

A DBSession service object represents an individual database session (a unique connection to a database). Unlike the DBResourceMgr service object, this service object also includes the user name and password to use for the connection, and the name of the specific database to connect to, as shown in [Figure 11-3](#):

**Figure 11-3** DBSession Service Object Database Tab Page

The database name for a DBSession service object is the specific database that you wish to access. When you create the service object, you can specify the specific database for which you wish to start a session. If you do not specify the database name when you create the service object, you must do so in the Partition Workshop when you partition the project.

When you create a DBSession service object, you specify the user name and user password that will be used for each database session. Because any number of end users will use the DBSession service object to access the database, you should use an application user name such as "Training," rather than a private user name, such as "Antoinette." This application name should have all the database privileges required by the application.

The fact that you must provide the user name and password as part of the DBSession service object definition may raise database security issues. However, the benefit of using a DBSession service object over a DBResourceMgr service object is that, with a DBSession service object, you can load balance the individual sessions for a large number of clients, which uses fewer database resources and provides improved performance.

## Setting Properties of Service Objects

When you create a new service object in the Project Workshop you give it certain properties; these properties are described in this chapter. You can modify some of these properties later on, under certain circumstances. For example, you can modify properties for the current configuration only in the Partition Workshop, and when you define a reference partition, you must specify an environment search path that is unique for that reference partition.

Note that changes to a service object that you make in the Partition Workshop apply only to the current configuration. The changes do not affect the original service object's definition.

## Class Runtime Properties

The class for a service object must allow the distributed class runtime property. At runtime, when iPlanet UDS creates the service object, it automatically sets the `IsAnchored` attribute to `TRUE`.

Although service objects are environment visible by default (that is, accessible by all clients), they should rarely be created as shared objects. In most cases you should not allow the `Shared` class runtime property for the service object's class, nor should you set the corresponding attribute, `IsShared`, to `TRUE`. If you make a service object a shared object, you cause it to be single-threaded, and thereby reduce performance. See [“The sharing provided by an environment-visible service object is different from the `Shared` runtime property for a class.” on page 401.](#)

## Service Object Properties

In addition to the class runtime properties, service objects also have another set of properties. These service object properties are shown in [Figure 11-1 on page 395](#), summarized in the table below, and described in the next sections of this chapter:

Property	Description
Visibility	Determines the scope of the audience served by a service object. Choices are <code>Environment</code> or <code>User</code> visibility. See <a href="#">“Setting Service Object Visibility” on page 399.</a>
Dialog Duration	Determines the interval over which a service object retains its connection with a caller and maintains state information for the caller. Depending on the type of service object, choices are: <code>Message</code> , <code>Transaction</code> , <code>Session</code> . See <a href="#">“Assigning a Dialog Duration to Service Objects” on page 402.</a>
Failover	If toggled ( <code>ON</code> ), then the service object is replicated, so that one or more backup service objects are available if the one in use becomes unavailable. Only one service object is in use at any given time. See <a href="#">“Providing Failover” on page 413.</a>

Property	Description
Load Balancing	If toggled (ON), then the service object is replicated, so that multiple service objects can be used at one time to accommodate many simultaneous users. iPlanet UDS creates a router to distribute connections among the available replicates. See <a href="#">“Providing Load Balancing” on page 420</a> .
Environment Search Path	If multiple iPlanet UDS environments are connected, allows service objects in other environments to be used in the current environment. See <a href="#">“Using the Environment Search Path” on page 432</a> .

You may need to modify these properties for a given service object, particularly if the service object is replicated for failover and/or load balancing, and is used by multiple applications. To modify anything other than the environment search path for a service object that is in a reference partition, you must open the Service Object Properties dialog from the configuration for the project that originally defined the service object.

## Setting Service Object Visibility

The *visibility* of a service object determines the scope of the audience served by a service object. A service object is defined with one of two possible visibility settings:

Visibility	Description
Environment	The service object is accessible to all users and services. This is the default.
User	A separate, private copy of the service object is created for each user or service requiring the service.

By default, a service object is environment visible. It can be accessed by all the users and services in the environment. The application contains only one copy of the service object, and any changes made to it are visible to any user or service in the application.

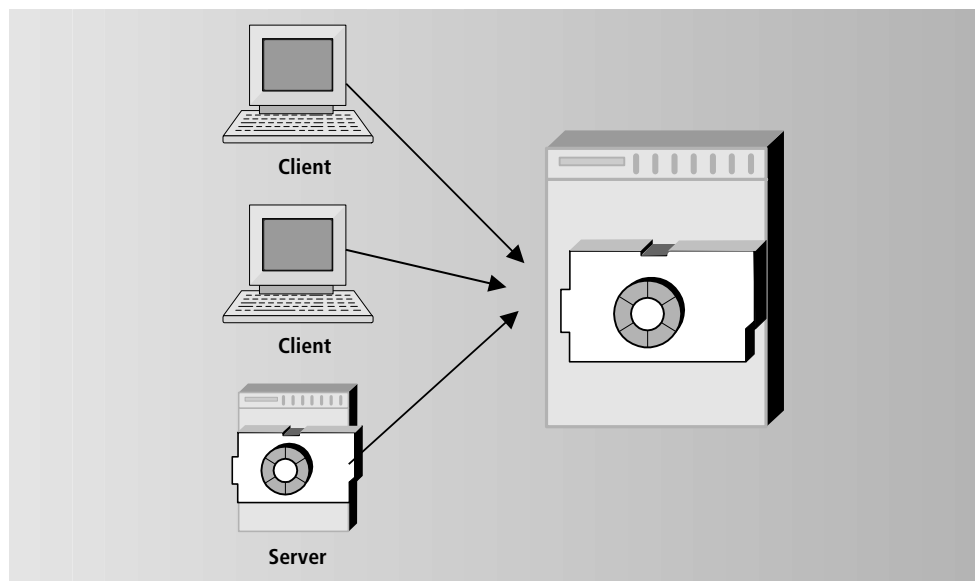
If you modify a service object to be user visible, it is not shared. Instead, a separate service object is created for each user of the service.

## Environment Visibility

Environment visibility is appropriate when a service object is a shared business service, and users must share application data. Because the end users all access the same service object, each change made to the object is automatically visible to all users.

For example, in the Art Auction application, the AuctionService service object is environment visible. This allows all Art Auction users to see the bids made by the other users, as illustrated in [Figure 11-4](#).

**Figure 11-4** Environment-Visible Service Object



Environment visibility provides the following advantages:

- efficient use of system resources

Because multiple clients can share a single service, your application scales better than if you had to provide a new server process for every single client.

- customized load balancing



When clients need more resources than a single service object can provide, you can replicate the shared service for load balancing. Using load balancing for environment-visible service objects allows you to provide exactly the number of replicates of the service you need. You can customize the application for each deployment environment by changing the number of replicates, and make continued adjustments as usage patterns change.

The sharing provided by an environment-visible service object is different from the Shared runtime property for a class.

Often you only require the *services* provided by the service object to be “shared”—that is, accessible to all users. In this case, you simply make the service object environment visible, and you do not allow the shared property for the class on which the service object is based.

However, if the service object contains *data* that should be shared by multiple users, you can allow this in two ways:

- Define individual attributes of the service object’s class as shared classes. This approach offers better performance for the service object because it does not serialize access to the service object.
- Set the Shared runtime property for the base class of the service object. This approach has the detriment of serializing access to the service object itself, since it is now a shared object and subject to mutex locking (see “[Class Runtime Properties](#)” on page 398).

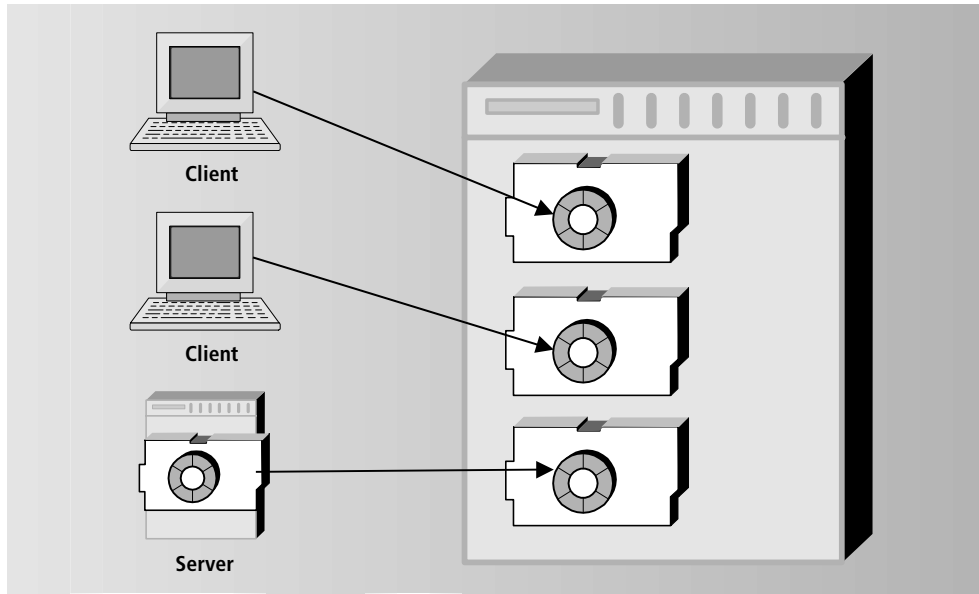
See “[Shared Objects](#)” on page 335 for information on the Shared runtime property.

## User Visibility

When you define a service object as user visible, a private copy of the service object is provided for each user or service that accesses the service object.

User visibility is appropriate when you want to distribute processing but you do not want to share the service object. Because each partition has its own copy of the object, a change made to the object by one partition will not be seen by any other partitions. The user visibility option is most useful when the service object represents a process that *cannot* be shared, such as a non-multi-threaded external service. [Figure 11-5](#) illustrates user visibility:

**Figure 11-5** User Visible Service Object



Depending upon your application's design, you can set DBResourceMgr service objects to have user visibility because the individual sessions started by the DBResourceMgr are private for the individual users.

DBSession service objects let you provide environment visibility for database access. DBSession service objects are more efficient than DBResourceMgr service objects because you do not need to run a separate session for each client.

You cannot use replication for failover or load balancing for a user-visible service object.

## Assigning a Dialog Duration to Service Objects

Every service object has a characteristic called dialog duration, whose value can be set to message, transaction, or session. The *dialog duration* is the interval over which a service object retains its connection with a particular caller after the first reference to the partition starts the connection. Any caller that makes requests from the service is bound to that particular service object for interval specified by the service's dialog duration.

For example, if a service object is using transaction dialog duration and is also replicated for load balancing, the transaction dialog duration guarantees that every caller will interact with only one service object replicate for the duration of the transaction. iPlanet UDS will never switch to another replicate in the middle of a transaction.

The dialog duration for a service object is a critical aspect of the services provided by a service object. Any caller that uses the service object must take the service object's dialog duration into consideration. Therefore, you specify the dialog duration as part of the original service object definition using the Project Workshop. And, unlike other service object properties, you cannot change the dialog duration setting from the Partition Workshop.

## Dialog Duration and State Information

The primary determinant of the appropriate dialog duration for a given service object is the length of time the service object should retain state information. The dialog duration reflects the period of time that state information is stored in the service object. State information is information that needs to be maintained for a *single user* of the service. For example, when a phone call is received by a call tracking system, the particular call being received is stored as state information for the duration of the phone call. This way, all routing done to the phone call is done to that single call in the proper order.

Typically, state information is ephemeral, and is no longer relevant after the end of the caller's interaction with the service. For example, after the call is disconnected, data resulting from the call may be stored permanently, but the call itself is over, so it no longer needs to be tracked.

When you design a service, you should consider how long the service should store state information—that is, what is the appropriate dialog duration. For example, in the Auction application, the image service is implemented as a message duration service object because the service simply provides image data in response to a message and it is therefore not necessary to retain state information.

However, for a document control service, the state information about whether or not a particular writer is logged into the service needs to be tracked the entire time the writer is doing her work. Therefore, session duration is the logical dialog duration for this kind of service.

## Dialog Duration and Error Handling

A service object's error handling depends on the service object's dialog duration. If there is a failure in the service, each dialog duration provides a different behavior. For example, recovery from a failure in a message-duration image service is relatively simple; iPlanet UDS automatically retries the request for an image. All the program needs to do is to notify the user that he may need to wait. On the other hand, recovery from a failure in a transaction-duration service would require handling the `AbortException` and retrying the transaction.

The following table summarizes the three dialog duration settings for service objects.

Duration	Description
Message	Retains the connection for the duration of a single method invocation. The service object does not retain state information beyond one method invocation. This is available only for TOOL class service objects. You can use message duration for load balancing or failover.
Transaction	Retains the connection for the duration of a transaction. The service object retains state information only for the duration of the transaction. This is available only for TOOL class and DB session service objects for load balancing or failover.
Session	Retains the connection for the entire run of the application. This is available for all service objects for failover replication, but not for load balancing. This is the default dialog duration for all service objects.

Note that if you wish to provide load balancing for a service object, you must specify either message dialog or transaction duration. See [“Relationship between Dialog Duration and Load Balancing” on page 423](#) for information on the relationship between dialog duration and load balancing.

See [“Relationship between Dialog Duration and Failover” on page 419](#) for information on the relationship between dialog duration and failover.

The following sections provide additional information about each of the dialog durations. Each section describes how the dialog duration affects state information, error handling, transactions, and events. The following table provides a brief overview of the affect of dialog duration on transactions, events, failover, and load balancing.

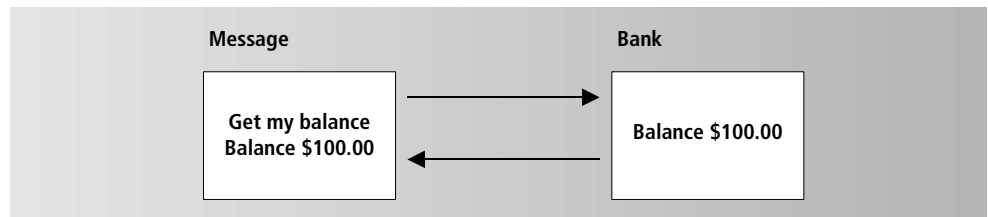
Dialog Duration	Transaction Propagated?	Uses Events?	Failover Automatic?	Load Balancing?
Message	No	No	Yes	Yes
Transaction	Yes	No	No	Yes
Session	Yes	Yes	No	No

## Message Duration Service Objects

With a message-duration service object, iPlanet UDS guarantees the connection to the remote service object only for the length of a single method invocation on the service object.

Figure 11-6 illustrates message dialog duration:

**Figure 11-6** Message Dialog Duration



An important advantage of message duration is that it is easy to provide load balancing and failover. For example, the ImageService service object does not retain state information, so switching over to a replicate for failover would not require additional programming to transfer state information.

When writing methods for any service, it is generally more efficient to write fewer methods that individually perform more processing, rather than more methods that individually perform less processing. Writing fewer methods is especially important for message dialog duration. Each method invocation to the remote service object is a “message,” so limiting the number of method invocations

improves distributed performance by decreasing network traffic. Note that a single method for the service can include an entire transaction or the logic to perform an entire complex procedure, and you can use multitasking to perform multiple tasks concurrently within the same method.

## State Information for Message Duration

Message-duration service objects cannot store shared state information beyond a single method invocation. State information is not stored because there is no guarantee that the caller will connect to the same service object for the next method invocation. For example, if load balancing is turned on for the service, one method invocation from a caller to the service could go to the first replicate, while the subsequent method invocation could go to the second replicate.

If you must share state information between the replicates, you can store it outside the service, in a database, for example. Because database information is a single source, the data is consistent, no matter which replicate accesses its information.

## Error Handling for Message Duration

If a message-duration service object fails during a method invocation, iPlanet UDS tries to re-establish a connection to any partition containing a replicate of that service object. Establishing failover servers is very useful for providing automatic recovery in these cases. If another replicate of the service is available, iPlanet UDS connects to it and re-invokes the method, as if it was still running on the original service. The program continues to run, and iPlanet UDS does not raise any exceptions. If iPlanet UDS cannot make a new connection, iPlanet UDS raises the `DistributedAccessException`, which you can handle in your TOOL code. See the Framework Library online Help for information about `DistributedAccessException`.

If a message-duration service object fails between method invocations, iPlanet UDS does *not* raise an exception.

## Transactions and Message Duration

For message-duration service objects, transactions in the caller are *not* propagated to the service. If you are in a transaction and invoke a method on a message-duration dialog service object, the message to the service object is considered *outside the transaction*. If the service fails while executing the method, iPlanet UDS does not automatically abort the transaction.

If the caller needs to start a transaction on the message-duration service object, the caller can invoke the method using the **start task** statement with the **independent** option. This version of the **start task** statement starts an independent transaction on the service object, which the caller can then handle appropriately. The caller cannot, however, start a nested or dependent transaction on the service object. See the *TOOL Reference Manual* for information about the **start task** statement.

## Events for Message Duration

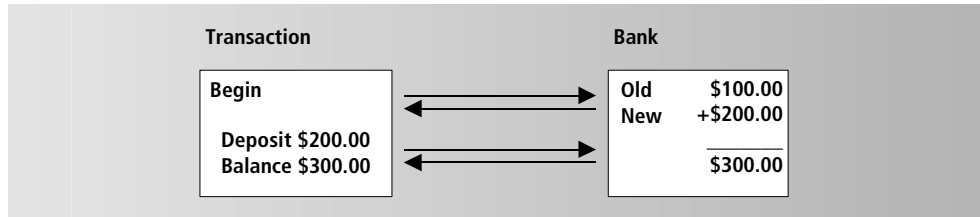
When the service object has message dialog duration, the caller should *not* register to receive events from the service or post events on the service. The connection to the service object is for one message only. Therefore, after an event is registered for one replicate of the service, there is no guarantee that the same replicate will post the event. Likewise, after an event is posted on one replicate of the service, there is no guarantee that the same replicate will handle the event.

While events might work for message-duration services when there is no replication for load balancing or failover, it is risky to count on this. As the programmer, you cannot control whether replication will later be turned on during the configuration or installation stages. Therefore, we recommend that you do not use events to interact with message-duration services. Instead, you should use the return value and output parameters of the method to return information or notifications.

## Transaction Duration Service Objects

With a transaction-duration service object, iPlanet UDS maintains the connection to the remote service object for the length of an iPlanet UDS transaction. The first method invocation on the remote service object after the start of the transaction initiates the connection to the service object, and the connection is retained until it completes or aborts. See the *TOOL Reference Manual* for information about using TOOL transactions.

**Figure 11-7** illustrates transaction dialog duration:

**Figure 11-7** Transaction Dialog Duration

## State Information for Transaction Duration

Transaction-duration service objects maintain state information for the duration of the entire transaction.

For example, a phone call tracking system might route an incoming phone call from initial screening to a dispatcher. The transaction begins when the phone is answered, and the transaction ends when the call has been properly handled. The state information about the particular phone call must be retained for the entire transaction, or, as long as the caller is on the line.

## Error Handling for Transaction Duration

If a transaction-duration service fails during a transaction, iPlanet UDS raises the `AbortException`, which provides a reason code for the failure and identifies the originator. You provide the same error handling as for a local transaction. If you are using failover or load balancing, you must write the code to retry the transaction (perhaps by enclosing your transaction in a loop) on another replicate.

For each retry, iPlanet UDS attempts to reestablish communication with any replicate of the service object, even if the new connection is on another partition. If the new connection succeeds, a new transaction starts with the new connection, and processing continues normally. If the new connection fails, iPlanet UDS raises a `DistributedAccessException`, from which you usually cannot recover. In some cases, however, you can handle the exception and then use a `Timer` object to wait a bit to see if the networks and servers recover. You can also use the `AbortEvent` event to get immediate notification when the service fails.

See the Framework Library online Help for information on the `AbortException`, `DistributedAccessException`, `Timer`, and `AbortEvent` classes.

If a transaction-duration service object fails and then starts up between transactions, iPlanet UDS does *not* raise an exception.



## Transactions and Transaction Duration

For transaction-duration service objects, transactions in the caller *are* propagated to the service. If you are in a transaction and invoke a method on a transaction-duration service object, the message to the service object is considered *part of the transaction*. So, if the service fails while executing the method, iPlanet UDS aborts the transaction.

As described under error handling above, when a transaction-duration service fails during a transaction, iPlanet UDS raises an `AbortException`. You must write the code to retry the transaction (perhaps by enclosing your transaction in a loop).

## Events for Transaction Duration

A caller should *not* register to receive events from the service or post events on a transaction dialog duration service.

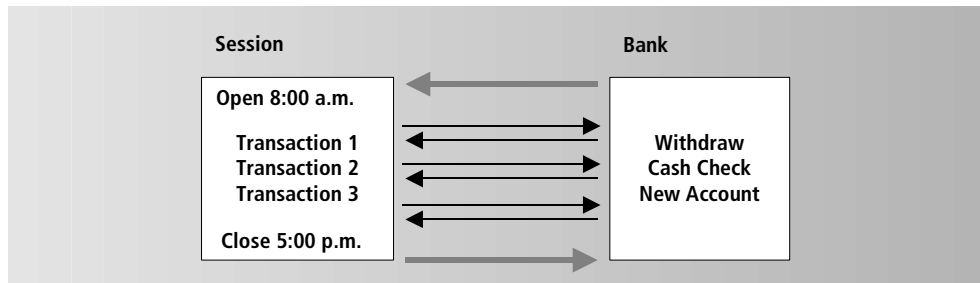
A connection to a transaction-duration service object is for one transaction only. So, after an event is registered for one replicate, there is no guarantee that the same replicate will post the event. Likewise, after an event is posted on one replicate of the service, there is no guarantee that the same replicate will handle the event.

While events might work for transaction-duration services when there is no load balancing or failover, it is risky to count on this. As the programmer, you cannot control over whether replication will later be turned on during the configuration or installation stages. Therefore, we recommend that you do not use events to interact with transaction-duration services. Instead you should use the return value and output parameters on the methods in the transaction to return information or notifications.

## Session Duration Service Objects

With a session-duration service object, iPlanet UDS guarantees a connection to the remote service object for as long as it is needed. The session starts when the first method invocation initiates the connection to the service object, and the connection is retained until the application completes or you use the `ReleaseConnection` method on the `DistObjectMgr` class (described in the Framework Library online Help) to release the connection explicitly.

**Figure 11-8** illustrates session dialog duration:

**Figure 11-8** Session Dialog Duration

## State Information for Session Duration

Session-duration service objects maintain state information for the duration of the entire session.

For example, a document control service allows writers to collaborate on a multi-volume documentation set. To use the document control service, the writer must login to the service, which then tracks the writer's activities until she logs out. The state information about who is currently using the service must be retained as long as the writer is logged in.

## Error Handling for Session Duration

If a session-duration service object fails during a session, iPlanet UDS raises the `DistributedAccessException`, which provides a reason code for the failure. You can attempt to handle the failure in your TOOL code. However, you cannot rely on state information if you successfully re-establish a connection to the remote service object, because, if the service object is replicated, the connection might be made to a different partition. To get immediate notification of a failure in the service, you can use the `RemoteAccessEvent` on the `Object` class. See the Framework Library online Help for information about the `DistributedAccessException` and `RemoteAccessEvent` classes.

If a session-duration service object fails and starts up again between sessions, iPlanet UDS does *not* raise an exception.

## Transactions and Session Duration

For session-duration service objects, transactions in the caller *are* propagated to the service. If you are in a transaction and invoke a method on a session-duration dialog service object, the message to the service object is considered *part of the transaction*. So, if the service fails while executing the method, iPlanet UDS automatically aborts the transaction.

As described in the error handling section, when a session-duration service fails during a transaction, iPlanet UDS raises the `AbortException`. You must write the code to retry the transaction (perhaps by enclosing your transaction in a loop).

After the transaction ends, the session itself continues on uninterrupted.

### Events for Session Duration

You can use events to communicate between a caller and a session-duration service object. The caller can register for events posted by the service object any time during the session. The service object can register for events posted by callers any time during the session. Events are very useful for communicating with a session-duration service.

## Replicating Servers for Failover and Load Balancing

You can replicate service objects and their respective server partitions to enhance application reliability using failover and to improve application performance using load balancing.

For example, you implement failover by providing backup replicates of a server partition that take over processing when the primary server partition fails. Similarly, you can implement load balancing to distribute demand for a service among a number of replicates of the partition that provides that service. You can also replicate a server for both failover and load balancing at the same time.

With some restrictions, a service object can be designated as:

- non-replicated (the default)
- replicated for failover
- replicated for load balancing
- replicated for both failover and load balancing

## Replication for Failover

If a service object is marked for failover, then you install replicates of the server partition on some number of nodes in an environment. If the primary server partition fails, then iPlanet UDS directs any requests for that service to one of the other running replicates of the server partition. For detailed information about using Failover, turn to [“Providing Failover” on page 413](#).

## Replication for Load Balancing

If a service object is marked for load balancing, then you also install replicates of the server partition on some number of nodes in an environment. In addition, iPlanet UDS automatically creates a *router partition* for that service object. The router partition distributes requests for that service among the running replicates of the server partition that performs that service. For detailed information about using load balancing, turn to [“Providing Load Balancing” on page 420](#).

## Replication for Failover and Load Balancing

If a service object is marked for both failover and load balancing, then you replicate the server partition as in the load balancing case. However, the router partition that iPlanet UDS automatically creates for the service object can also be replicated. You replicate the router partition and install it on a number of nodes in an environment in the same way that you replicate the server partition. iPlanet UDS designates one of the router partitions as the *primary router*. If it should fail, another running replicate of the router partition can handle the routing.

When you install any replicated partition on a node in an environment, be it a server partition or a router partition, you can specify how many replicates of that partition are “enabled” or started up on that node, when the application starts. Installed partitions that are designated as “disabled” can be started manually to provide special backup under unusual conditions.

# Providing Failover

*Failover* means providing backup service objects (called service object *replicates*) to be used if a service object fails. Failover provides built-in fault tolerance for an application. iPlanet UDS provides two kinds of failover:

**Local failover** To provide local failover within the current environment, you can create any number of replicates of the partition that contains the service object. If the primary service object should fail, iPlanet UDS automatically switches over to the secondary replicate partition. If the secondary replicate fails, iPlanet UDS uses the next replicate if there is one, and so on.

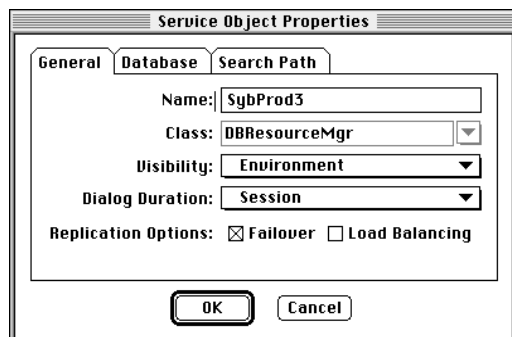
**Cross-environment failover** When the deployment environment is connected to other environments, you can specify that if the primary service object in the current environment fails, iPlanet UDS searches in other environments for another active service object to use for failover. You do this by providing an environment search path for iPlanet UDS that specifies which environments, in which order, to search when primary service object fails.

The following two sections describe how to provide local and cross-environment failover respectively. You can use both types of failover in combination; see [“Combining Local and Cross-Environment Failover” on page 418](#).

## Enabling Failover

To request local failover for a service object, switch on the Failover toggle in the service object definition. You can do this in the original service object definition in the Project Workshop (as shown in [Figure 11-9](#)) or for a particular configuration using the Partition Workshop.

**Figure 11-9** Service Object Dialog with Failover



When you turn on failover for the service object, the Partition Workshop assigns the service object to a replicated partition. You can then assign the replicated partition to any number of suitable nodes in the environment. If you wish, you can also assign the partition any number of times on any individual node. See *A Guide to the iPlanet UDS Workshops* for further information about assigning replicated partitions to nodes.

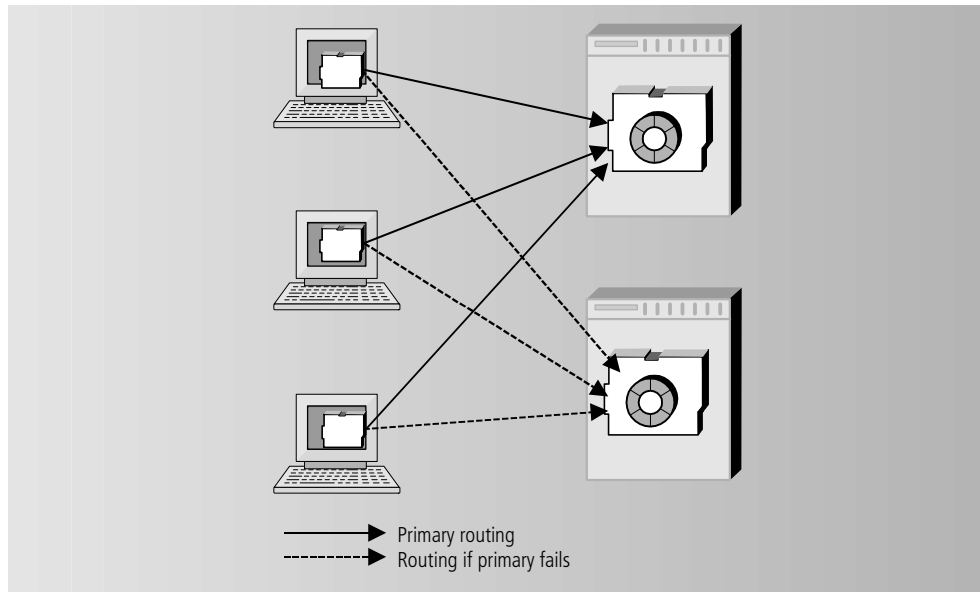
## Failover in the Local Environment

You can replicate service objects in the local environment to provide for failover in the case of either software or hardware failure (or both).

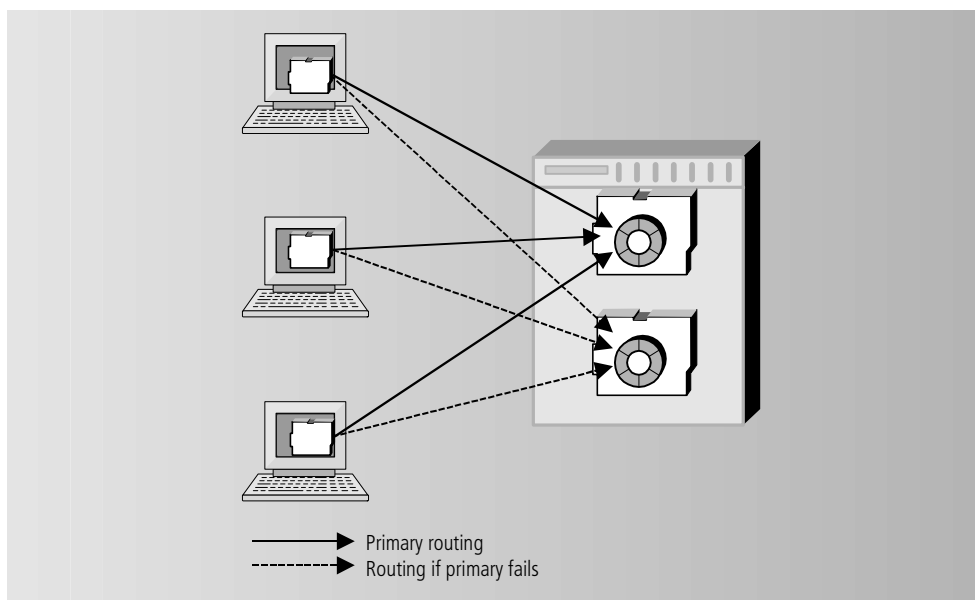
To provide failover for hardware failure in the current environment, you install one or more replicates on different nodes in the environment, as shown in

**Figure 11-10:**

**Figure 11-10** Hardware Failover



To provide failover for software failure in the current environment, you install one or more replicates on the same node, as shown in **Figure 11-11**.

**Figure 11-11** Software Failover

When a service object is defined with failover and the primary service object fails, iPlanet UDS automatically switches over to the secondary replicate. If the secondary replicate fails, iPlanet UDS uses the next replicate if there is one. (See [“Relationship between Dialog Duration and Failover” on page 419](#) for information about when failover is automatic and when you need to provide code to perform retries.)

You can provide any number of failover replicates that you wish. The replicates can be on the same node, on different nodes, or any combination of the two.

For local failover, you have no control over the order in which replicates are used for failover. One replicate is used for all processing until it fails. When the first replicate fails, iPlanet UDS uses a second replicate for all processing. You cannot specify which of the replicates should be used first, and which should be used for backup.

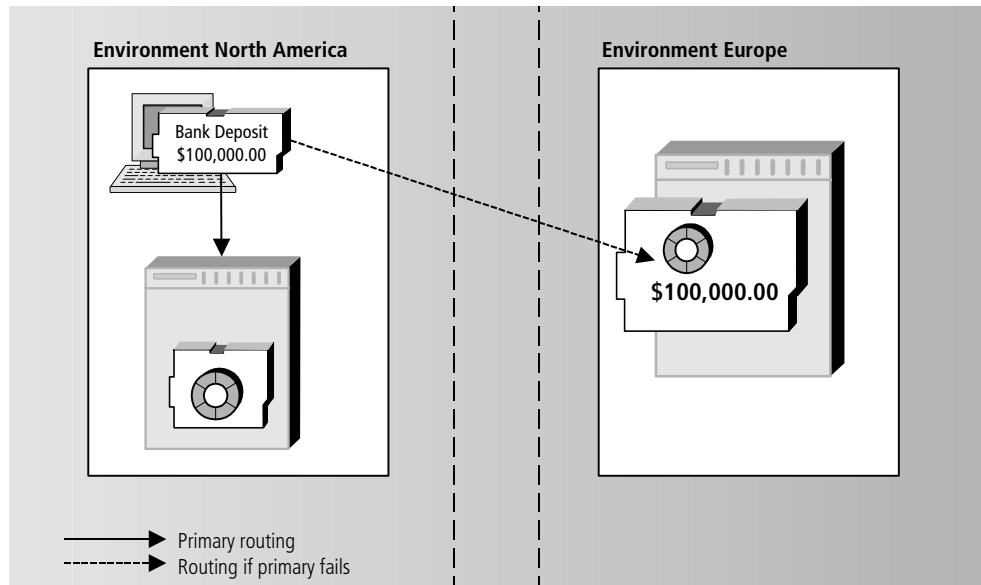
Because only one replicate at a time is used for all processing, failover does *not* provide parallel processing. When a service object is replicated for failover, iPlanet UDS maintains a connection to only one replicate at any time. The others are merely in place in case of failure. See [“Providing Load Balancing” on page 420](#) for information about parallel processing.

## Cross-Environment Failover

If deployment environments are connected, you can use cross-environment failover to provide a backup service object in a different environment to be used if the primary service object in the current environment fails. Cross-environment failover is useful when an application accesses a critical service that *must* be available.

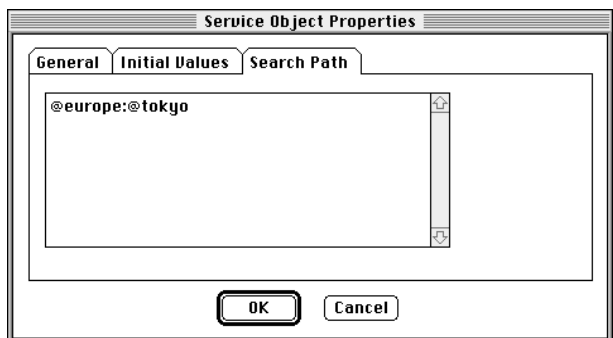
For example, [Figure 11-12](#) shows a banking application running in the North America environment that can access an equivalent service in the Europe environment when its own service fails. During failover the application in North America can continue to run, although in a degraded fashion, until its local environment is fully functional again.

**Figure 11-12** Cross-Environment Failover



To provide cross-environment failover, you specify an environment search path for the service object, using the Service Object dialog in the Partition Workshop. As shown in [Figure 11-13](#), the path should list one or more connected environments that contain service objects that can be used for failover. See [“Using the Environment Search Path” on page 432](#) for more details.



**Figure 11-13** Service Object Properties Environment Search Path Tab Page

At runtime, when a caller accesses the service object, iPlanet UDS checks the environment search path. It searches the first environment in the search path for the service object (therefore, be sure to include the current environment as the first environment in the search path). If the service object in the first environment is not available, iPlanet UDS searches the next environment in the search path. iPlanet UDS continues to search the environments in the path until it finds an accessible service for failover or it runs out of environments. The order of the environments in the path lets you control which backup service is used first, second, and so on.

See [“Relationship between Dialog Duration and Failover” on page 419](#) for information about when failover is automatic and when you need to provide code to perform retries.

## Deploying Applications with Cross Environment Failover

You have several options for deploying applications that use cross-environment failover, including the following two general approaches:

**Deploy one complete application in every environment** If the complete application is used in all connected environments, you can simply deploy a complete application in every environment. The applications in each environment can then use the other service objects for failover. (You must set each environment search path appropriately.)

If one environments is used solely for a backup service, the system manager can start up just the server partition for the application in that environment. Therefore, you do not need to run the entire application in every deployment environment. See the *iPlanet UDS System Management Guide* for more information.

**Create two separate applications (server application and client application)**

The server application would just define the server that needs cross-environment failover. You could then deploy the server application in all the environments where you need it, including environments you want to use just for backup.

The client application would define the client part of the application, and would provide access to the server through a reference partition. The environment search path for the service object in the reference partition would contain the list of the environments to search for the failover service. (See [“Sharing Service Objects Between Applications”](#) on page 426 for information on reference partitions).

## Combining Local and Cross-Environment Failover

The previous two sections described how to provide local failover or cross-environment failover. You can also use both local and cross-environment failover. iPlanet UDS uses the local failover replicates before searching in the connected environments.

► **To combine local and cross-environment failover**

1. In the Project Workshop, open the Service Object properties dialog for the service object. Turn on the Failover toggle to request local failover.
2. In the Partition Workshop, assign the replicated partition to the appropriate nodes in the current environment and specify their replication count.
3. In the Partition Workshop, open the Service Object properties dialog for the service object. Enter the environment search path for the service object. Make sure the first environment in the path is the current environment.

At runtime, when the caller accesses the service object, iPlanet UDS checks the first environment in the search path for any active replicate. If for any reason, all of the failover replicates in that environment are not available, iPlanet UDS searches the next environment in the search path. iPlanet UDS continues searching environments in the path until it finds an accessible service for failover or it runs out of environments.

The following section provides information about when failover is automatic and when you need to provide code to perform retries, based on the service object’s dialog duration.

## Relationship between Dialog Duration and Failover

iPlanet UDS provides automatic failover only for service objects with message dialog duration; for service objects with message or transaction dialog duration, you must write the code to retry the method upon receiving the expected exception, as described below.

iPlanet UDS performs failover automatically for message-duration service objects. If a message-duration service object fails during a method invocation, iPlanet UDS automatically switches to the failover replicate without notifying the application. You do not need to provide any code to retry the message. iPlanet UDS automatically tries all the replicates, as necessary.

If a message-duration service object fails and no failover replicates are available, iPlanet UDS raises the `DistributedAccessException`. See [“Error Handling for Message Duration” on page 406](#) for information.

If a message-duration service fails between messages, iPlanet UDS simply uses a failover replicate on next access and does not notify the application.

iPlanet UDS does not perform failover automatically for transaction-duration service objects. You must provide the code to retry the transaction. If a transaction-duration service fails during a transaction, iPlanet UDS raises the `AbortException`. You must provide the code to handle the `AbortException` and retry the transaction (perhaps by enclosing your transaction in a loop). See [“Error Handling for Transaction Duration” on page 408](#) for information.

For each retry, iPlanet UDS attempts to reestablish communication with any replicate of the service object, even if the new connection is on another partition. If the new connection succeeds, a new transaction starts with the new connection, and processing continues normally. If the new connection fails, iPlanet UDS raises a `DistributedAccessException`.

If a transaction-duration service fails between transactions, iPlanet UDS simply uses a failover replicate on next access and does not notify the application.

iPlanet UDS does not perform failover automatically for session-duration service objects. You must provide the code to retry accessing the service. If a session-duration service object fails during a session, iPlanet UDS raises the `DistributedAccessException`. Your code must handle the `DistributedAccessException` and reconnect with a failover replicate by invoking a method on the service again. See [“Error Handling for Session Duration” on page 410](#) for information.

For each retry, iPlanet UDS attempts to reestablish communication with any replicate of the service object, even if the new connection is on another partition. If the new connection succeeds, a new session starts with the new connection, and processing continues normally. If the new connection fails, iPlanet UDS raises a `DistributedAccessException`.

If a session-duration service fails between sessions, iPlanet UDS simply uses a failover replicate on next access and does not notify the application.

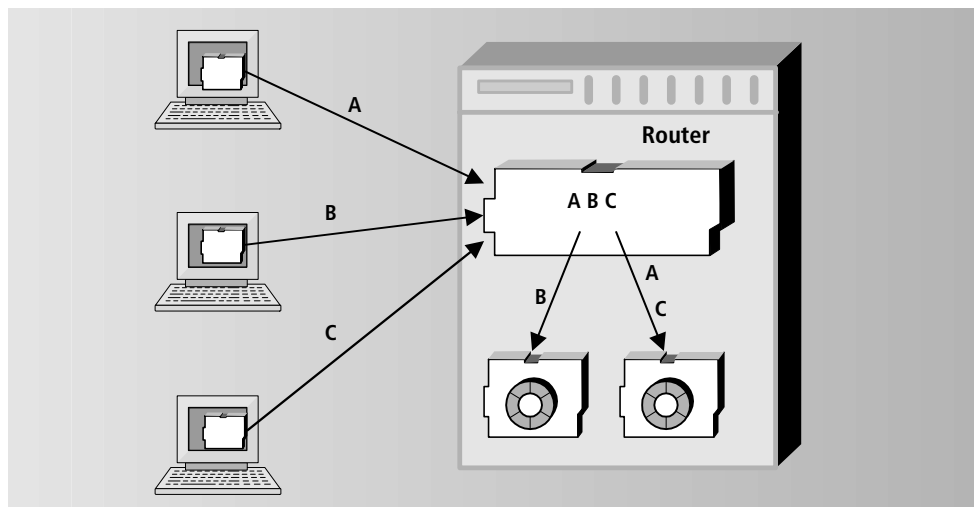
## Providing Load Balancing

Load balancing is the best way to scale an application to support large numbers of clients that need a common service. *Load balancing* means using multiple replicates of a service object to provide simultaneous access for several clients and increase throughput. When you use load balancing, iPlanet UDS automatically creates a *router partition* that coordinates all connections to the replicates, distributing connection requests for best performance.

You can load balance a service object using two general techniques.

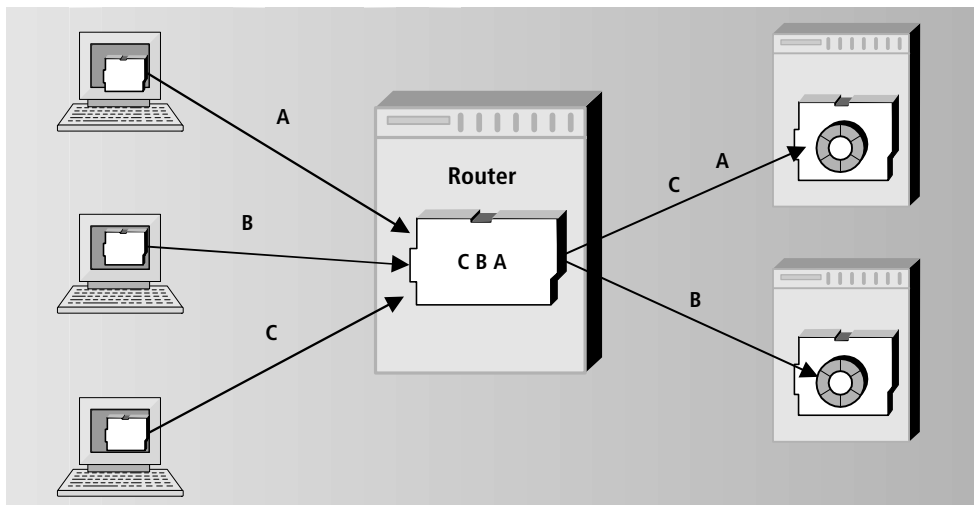
**Run replicated service objects on a single machine** This technique allows you to pool resources so as to use hardware more efficiently. As shown in [Figure 11-14](#), requests from the clients to the server are routed to different partitions on the same node.

**Figure 11-14** Load Balancing on a Single Node



**Run replicated service objects on multiple machines** This technique lets you improve performance by providing parallel processing on different machines. In [Figure 11-15](#) service object replicates are installed on two machines and the router on a third machine. The router sends requests from the clients to different partitions on different nodes:

**Figure 11-15** Load Balancing on Multiple Nodes



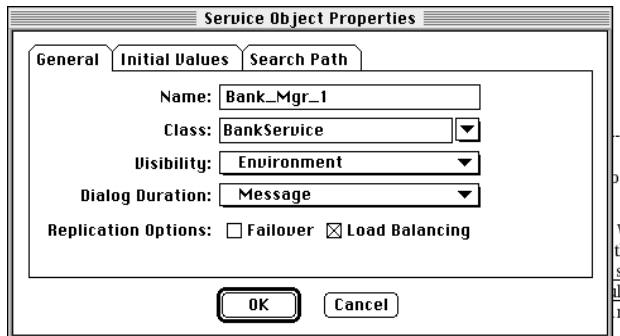
You can combine these two techniques, so that multiple machines have multiple replicates of the same service object.

## Enabling Load Balancing

You can turn on load balancing for any service object that uses message or transaction dialog duration. See [“Relationship between Dialog Duration and Load Balancing” on page 423](#) for information on the relationship between dialog duration and load balancing.

► **To turn on load balancing for a service object**

1. To modify the original service object definition, use the Service Object dialog in the Project Workshop. Turn on the Load Balance toggle, as shown below:



2. Alternatively, to enable load balancing for a service object for a particular configuration, you can use the Logical Partitioning (left) side of the Partition Workshop.

In both cases, the Partition Workshop assigns the service object to a replicated partition and automatically creates the router partition (see [“The Router Partition” on page 424](#)).

3. Assign the replicated partition to one or more suitable nodes in the environment. You can also assign the partition any number of times on any individual node.

In this step, described in the next section, you set the number of replicates on a node-by-node basis.

## Setting the Number of Replicates

You can specify any number of replicates of a service object for load balancing; if the partition (service object) is specified for auto-start, that number of replicates will auto-start. The optimal number of replicates depends on the particular environment. You may need to balance the number of concurrent requests to the service (the more replicates, the better the response) with the physical resources of the particular environment (how many replicates can be supported?).

► **To specify the number of replicates for a particular configuration**

1. Open the Partition Workshop.
2. From the Assigned Partition (right) side of the workshop, open the Properties Dialog for the partition that contains the service object that you wish to replicate.
3. In the Replication Count field enter the number of replicated partitions to start up.

The system manager can tune performance at any time by changing the number of replicates for a particular installation.

### Relationship between Dialog Duration and Load Balancing

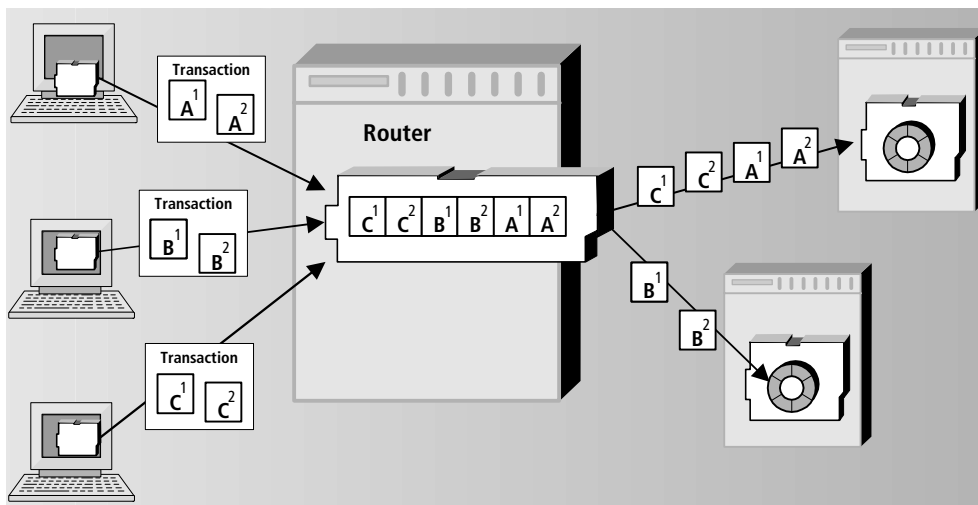
For a message-duration service object, the router routes each method invocation individually. Therefore, every method invoked on the service may be routed to a different replicate partition.

If a message-duration service object fails during a message, the router simply re-routes the message to one of the remaining replicates without notifying the program. This provides automatic failover.

For a transaction-duration service object, the router routes an entire transaction to a single replicate. Load balancing with transaction duration guarantees that all messages for one transaction are sent to the same replicate. As described under [“State Information for Transaction Duration” on page 408](#), the service object maintains the state information for duration of the transaction.

[Figure 11-16](#) illustrates load balancing for a transaction-duration service object:

**Figure 11-16** Load Balancing with Transaction Dialog Duration



If a transaction-duration service object fails during a transaction, iPlanet UDS raises the `AbortException`. You must write the code to handle the `AbortException` and retry the transaction (perhaps by enclosing your transaction in a loop). The router will automatically route the transaction to an active replicate. See [“Error Handling for Transaction Duration” on page 408](#) for information.

## The Router Partition

The router partition for a load-balanced service object provides a single router that coordinates all requests to the service. The router queues all requests for service and directs each request to a service object replicate using a round robin algorithm. “Round robin” means that the first request goes to the first replicate, the second request to the second replicate, and so on, back to the first replicate.

The router routes each request as it reaches the top of the queue; it does not wait until one request completes processing (that is, a server becomes available) before routing the next request. This way, processing is distributed evenly across the partitions, regardless of machine resources. If your resources vary from machine to machine, you can assign more replicates of the partition to one machine and fewer replicates to another.



The connection between the router and an individual replicate depends on the dialog duration for the service object. For a message-duration service object the connection is for the duration of the message and for a transaction-duration service object the connection is for the duration of the transaction.

You can assign router partitions to the same node where one or more replicates is installed, or to a different node. However, you should not place any service objects on the router partition, because you do not want to impede its performance.

You can replicate the router for failover, similar to replicating a service object; see [“Failover for the Router”](#) below. If the router fails and there are no failover replicates, iPlanet UDS raises the `DistributedAccessException`.

## Single-Threaded and Multi-Threaded Routers

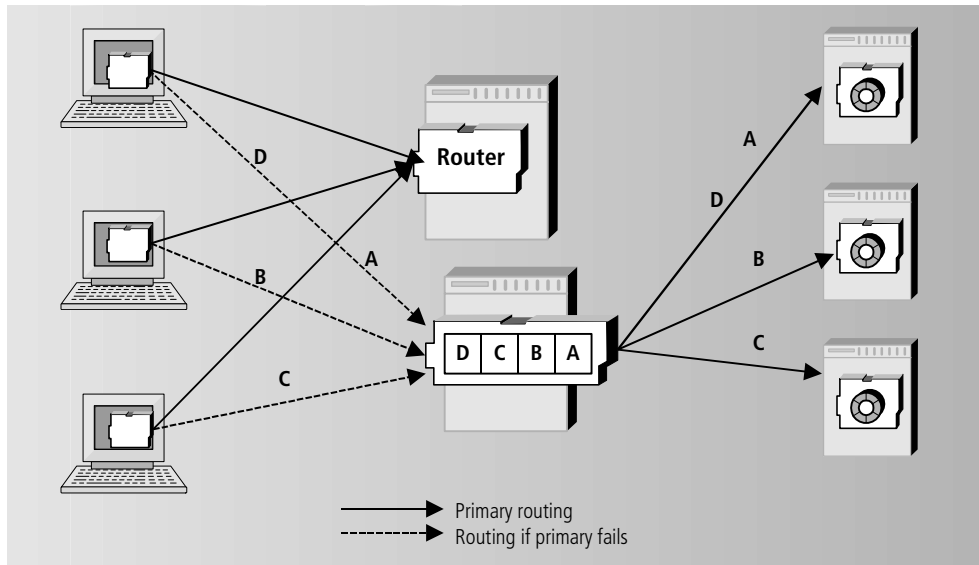
Without load balancing, requests from more than one caller can be processed concurrently within a single partition (*multi-threading*). However, if multiple threads each need to access a shared resource that does not allow multi-threading (a database, for example), the first thread to access the shared resource blocks the second thread until the first thread completes.

When load balancing is turned on, the router controls the access to the partitions so that only one request for service is sent at a time (*single-threading*). Therefore, the requests cannot conflict with each other, which allows concurrent access to the shared resource. Load balancing access to a single-threaded, shared resource provides much more even response time.

If you have a TOOL class service object that accesses a database or other single-threaded resource, it is recommended that you use load balancing to provide concurrent access to the resource. For information on multi-threaded access to databases, refer to the *Accessing Databases* manual.

## Failover for the Router

Because the router is critical for load balancing, you should also replicate the router for failover. Then, if the primary router fails, a backup router automatically reconnects to all the replicates of the service object, as shown in [Figure 11-17](#).

**Figure 11-17** Router with Failover

To provide failover for the router, you must turn on both the Failover and Load Balance toggles in the service object definition. You can do this either in the original service object definition in the Project Workshop or for a particular configuration using the Partition Workshop.

When you turn on failover for the router, the Partition Workshop assigns the router to a replicated partition. You can then assign the replicated router partition to any number of suitable nodes in the environment or any number of times on any individual node. You assign the router partition to a node the same any other replicated partition. See *A Guide to the iPlanet UDS Workshops* for additional information about assigning replicated partitions to nodes.

You must provide the same error recovery for the router as for any failed partition—dependent on the service object’s dialog duration. See [“Error Handling for Message Duration” on page 406](#) and [“Error Handling for Transaction Duration” on page 408](#) for more information.

## Sharing Service Objects Between Applications

Multiple applications running in the same environment can share a service object. Furthermore, applications running in different environments can share service objects, if the environments are connected.

To enable multiple applications to share a service object, you use reference partitions. First you must deploy the service object in one application, either a server or a client application—it makes no difference. Then, you create a *reference partition* in each of the other applications. A *reference partition* points to an existing service object that is deployed as part of another application; a reference partition does not contain the service object itself. The advantages of using reference partitions include:

**Modularity** When multiple applications share a service object, you need only create and manage a single service object. Without reference partitions, you would need to create more than one version of the service.

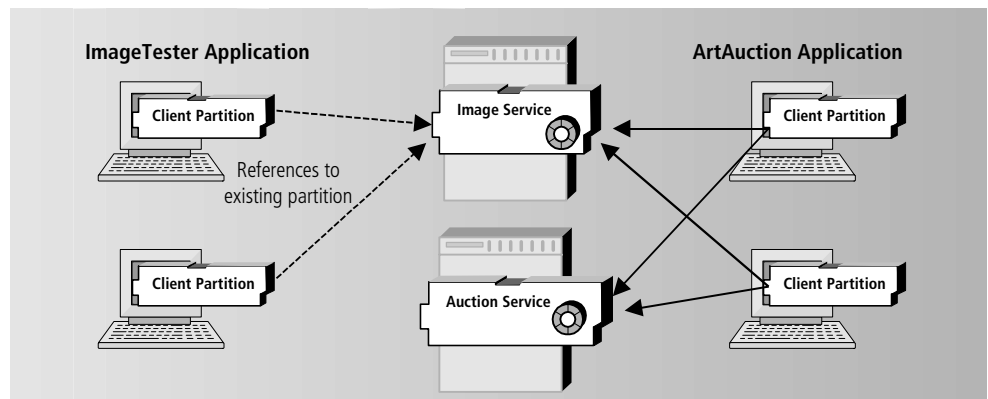
**Efficient use of resources** Only one service object needs to be running in order to provide services needed by multiple applications. Without reference partitions, the service object need to run within every application that needs it.

## Sharing a Service Object in a Single Environment

It is not uncommon for several applications in the same environment to require the same service. Using reference partitions, you can enable all applications to share a single service object.

For example, in [Figure 11-18](#) two applications, ImageTester and ArtAuction, both require the ImageServer.

**Figure 11-18** Reference Partition for Single Environment



Assume the ArtAuction application already provides the ImageService service object. Then the ImageTester project can include the project containing the ImageService as a supplier plan, and create a reference partition that points to the existing ImageService partition. Thus, both applications can share the existing service, avoiding the creation of a new instance of the service object.

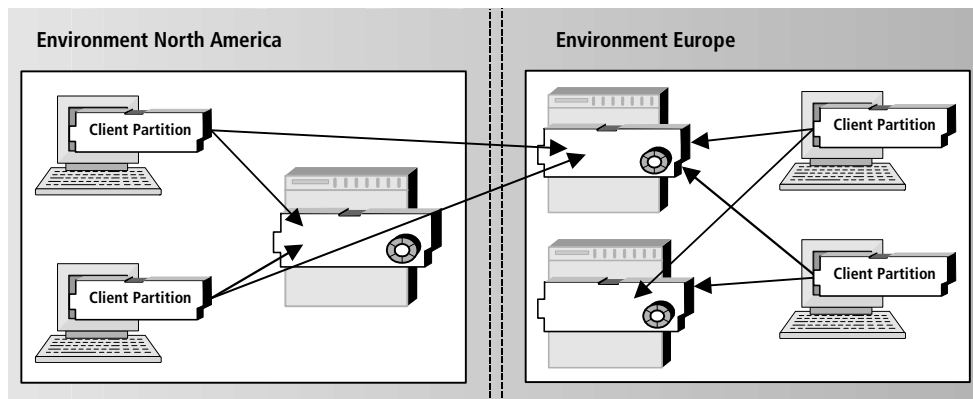
## Sharing a Service Object in Connected Environments

An iPlanet UDS system manager can connect multiple iPlanet UDS environments together, to gain advantages in performance, maintenance, and scaling. When the environment in which you are partitioning a project is connected to one or more environments, you can use service objects from those environments in the current configuration.

Sharing services is particularly useful when a particular service can run only in one environment. One example is a service that uses a specialized satellite feed or stock ticker that is accessed through a callout to an external service running in one location. Another example is a service that handles personnel data and that only resides in the headquarters environment, because that is where the personnel database is. All other applications installed in their own local environments must access the personnel database located at headquarters.

To allow applications in connected environments to share a specialized service, you use reference partitions. **Figure 11-19** illustrates two connected environments:

**Figure 11-19** Reference Partition for Connected Environment



## Including a Shared Service Object in an Application

The steps required to share a service object are summarized below and described in more detail in following sections.

► **To include a shared service object in an application**

1. Deploy the application that defines the service object that is to be shared.
2. If the service object is in a connected environment, you must make a distribution for the server application that contains it in the current environment (although you do not install it).
3. In the application that needs to access the deployed service object, include as a *supplier plan* the project from the deployed application that originally defined the service object.
4. Use the Partition Workshop to create a reference partition that points to the existing service object.

If the service is in a connected environment, the environment search path for the service object in the reference partition must specify the environment where the original server application that contains the service object is deployed.

### Deploying the Shared Service

Before you can include a shared service object in your application, the application that defines the service object must be deployed.

The application that defines the service for deployment can be a server application or a client application—it makes no difference. Often the application that defines the service for deployment is a server application that simply provides the service object definition. The other applications that will share the service define various user interfaces that enable end users to interact with the shared service. However, you could also implement the service as a client application that includes a small interface for testing and managing the service.

For example, assume you want to deploy a image service to be shared by various applications. You could define an ImageService project, which defines the image service itself. You could also define an ImageTester project, which provides a simple testing facility for the service. You could then include the ImageService project as a supplier to ImageTester, and configure and deploy ImageTester as a client application. Later on you could include the ImageService project as a supplier project for other applications.

## Making a Distribution for the Server Application in the Local Environment

If the service to be shared is in a connected environment, you must make a distribution for the service in the local environment:

### ► To make a distribution for the server application

1. In the local environment, partition the application that defines the service, and make a distribution for it.
2. Do *not*, however, install the application.

In this step you provide the local environment with both the logical and deployment definitions of the service. These definitions are needed to enable the application in the local environment to use the service in the connected environment. However, because the application is not installed in the local environment, iPlanet UDS will never attempt to connect to an instance of the service in the local environment.

## Including the Supplier Project

When you define an application that will reference an existing service, you must include the project that defines that service object as a supplier project for your main project. The supplier project that defines the service object must be exactly the same project that was originally used to deploy the service for the existing application. Even though you will not be deploying that service object from within the current application, iPlanet UDS needs the original service object definition to represent the existing service object within your new application.

The supplier project in the current application must match the project in the deployed application in the following aspects:

- the names of the projects must match
- the compatibility levels of the projects must match
- all runtime IDs must match

► **To transfer a project from one repository to another**

1. Start Fscript and use the command **ExportPlan**, with the **ids** option.

---

**CAUTION** You must use Fscript command **ExportPlan** to export the project *and* you must export with the **ids** option. The IDs are required to ensure the exported project is identical to its source, and you must use Fscript because the Export command in the Repository Workshop does not export with IDs.

---

2. Import the exported plan into a repository in the local environment.

For example, assume you are creating an Auction project that defines an application that needs to access the image service. If, as described in the previous section, you had previously deployed that image service as part of the ImageTester application, you would need to include the ImageService project as a supplier in the Auction project. You need to include the ImageService project because this project defines the service object you want to access. You do not need to include the ImageTester project because this project defines only the testing interface for the image service.

## Making the Reference Partition

When you partition an application in the Partition Workshop, iPlanet UDS creates new partitions containing all the service objects in the main project and all of its supplier projects. Therefore, the service object that you are planning to include in the reference partition will be on a new partition (or it may share a new partition with other service objects).

At this point, you must create the reference partition to point to the deployed service object, and move the service object from the new partition into the reference partition. This tells iPlanet UDS to use the service object that the reference partition points to, rather than creating a new instance of the service object.

See *A Guide to the iPlanet UDS Workshops* for information about making a reference partition using the Partition Workshop.

If your deployment environments are connected and a service that you need to access can run only in one of the environments, you can access that shared service from other environments by using a reference partition combined with an *environment search path*.

*In a reference partition to a connected environment, you specify an environment search path for the service object you need to access. The environment search path includes the name of the environment that contains the service. Then, whenever the service object is referenced, iPlanet UDS will use the service object in the specified environment. See “Using the Environment Search Path” for more information.*

When you have applications in multiple environments that need to use a specific service in a specific environment, you should deploy application that contains the service object in the environment in which it can run. Then, each application in the other environments can create a reference partition to the service object in that application. The search path from each application would *only* include the specific environment where that service is deployed. (It must *exclude* the local environment.)

## Using the Environment Search Path

You can use a service object that is in a different environment in two ways:

- You can share an existing service in a connected environment, rather than starting that same service in the current environment.

You do this by using a reference partition (described earlier under “[Sharing Service Objects Between Applications](#)” on page 426).

- You can provide a list of service objects in connected environments to be used for failover for a service object in the current environment.

You do this by specifying an environment search path for the service object (described under “[Cross-Environment Failover](#)” on page 416).

If your environment is not connected to other environments, you can only use services within the current environment. To see if a given environment is connected to any other environments, use the Environment Console or Escript. For more information, see the *iPlanet UDS System Management Guide*.



## Specifying an Environment Search Path

An *environment search path* for a service object is a list of connected environments in the order that iPlanet UDS should search for that service object. You can specify an environment search path at several levels. Depending upon where a search path is specified, iPlanet UDS uses the path to locate only the specified service object, or all service objects. You can specify the environment search path in the following ways:

- in the Environment Console (or Escript)

The iPlanet UDS system manager can provide a default search path for the environment as a whole. This search path is used for all service objects in the environment that do not have their own environment search paths. (Typically, you should not set the search path at this level—it is best to use the system default.)

- in the Project Workshop

You can provide a search path in the original service object definition. This search path overrides (that is, does not add to) the default search path for the environment, and it is used for the particular service object in all configurations that do not override it for the particular configuration.

- in the Partition Workshop

You can provide a search path for an individual service object on an assigned partition. This search path overrides (that is, does not add to) the search path specified in the Project Workshop, and it is used only for the particular service object on the particular assigned partition.

*A Guide to the iPlanet UDS Workshops* describes setting the path in the Project Workshop or the Partition Workshop. The *iPlanet UDS System Management Guide* describes setting the path in Econsole or Escript.

If no search path is specified at any of these levels, iPlanet UDS uses the system default search path. The system default search path searches only the current environment, regardless of whether or not the current environment is connected to other environments.

The system default search path is the following:

@ (a)

The @ means that only the current environment is searched, and the (a) means that all service objects are automatically started when necessary.

## Specifying Auto-Start for a Partition

By default, all partitions in an environment start up automatically. The iPlanet UDS system manager can change this default by modifying the default environment search path. If your iPlanet UDS system manager has done so, you can override his settings to specify that one or more partitions start automatically.

To specify that a partition auto-start, you use the environment search path option (a). This option requests that the associated service object in that environment be started automatically (and thus, the partition that contains the service object).

Even if your environment is not connected, you can use an environment search path to request auto-start for a particular service object. Simply enter the following search path in the Project Workshop for the service object that you want to auto-start:

@(a)

Because disabled partitions are not auto-started, if the environment search path specifies a disabled partition with the (a) option, that partition will not auto-start, and iPlanet UDS will look in the next entry in the environment search path for the service object.

# Advanced Options for Structuring Client Applications

This chapter explains how you can design iPlanet UDS applications that use the following features:

- applets
- nomadic clients

This chapter assumes that you have Framework Library online Help available.

## Writing Applications That Use the Launch Server and Applets

The AppletSupport library provides classes that allow you to write the following:

- applications that use the Launch Server to launch other iPlanet UDS applets or applications
- customized applications that let end users access the Launch Server

This section introduces the classes, methods, and attributes of the AppletSupport library. Reference information for the AppletSupport library is more thoroughly documented in Framework Library online Help.

The classes in the AppletSupport library are:

**AppletData** Represents a client application that can be launched using the LaunchService service object.

**LaunchMgr** Provides access to the features of the LaunchService service object, also defined in the AppletSupport library.

**AppletReleaseData** Contains information about a specific release of a client application.

**AppletRunInfo** Represents information about a client application that was started by the `LaunchService` service object and is currently running.

## Setting up the AppletSupport Library

To use the AppletSupport library, import the `apltsupp.pex` file in the `FORTE_ROOT/userapp/appletsu/cl0` directory into your repository.

When you write an application that uses any of the classes of the AppletSupport library, you need to include this library as a public plan in your workspace and include this library as a supplier plan for the application.

## Advantages of Using the AppletSupport Library

You can use the AppletSupport library to design applications that start other applications in the same `ftexec` process. You can start these applications using the `LaunchMgr.RunApplet` method.

---

**NOTE** Although this library is intended to support the use of applets as parts of applications, you can also start deployed applications that are not applets using the features of the AppletSupport library.

---

There are a number of advantages to starting other client applications from within your `TOOL` code using the `LaunchMgr.RunApplet` method:

- reduce the size of a client partition by dividing the functions among client applications (or applets), then launching these client applications using the Launch Server
- start other applications in the same process, which can be faster than starting the applications in different processes
- start, list, and shutdown launched client applications using `LaunchMgr` methods

Typically, when you write a complex client for an iPlanet UDS application, the client partition is large. If you design your application to reuse existing client partitions, you have the choice of using the following methods:

**LaunchMgr.RunApplet** Starts an installed or publicly-available applet or application in the same process as the main client application. When your application launches another client application using the `LaunchMgr.RunApplet` command, that application cannot communicate with the application that started it. However, the other application shuts down when the starting application shuts down.

**OperatingSystem.RunCommand** Starts an installed client partition in another process. The command specified for this method is not necessarily portable across platforms.

## Restrictions

There are a number of limitations that you should be aware of when you design an application that starts other applications and applets using the `AppletSupport` library. Applications or applets that have been started by a main client application *cannot* do the following:

- propagate events to the main client application, or to other applications or applets that were started by the main client application
- define and pass around references to anchored objects in other applications or applets
- share user-visible service objects between client partitions

## Building Applications by Starting Multiple Smaller Applications

This section discusses writing a client application that starts other applications using the `LaunchMgr.RunApplet` method. This sections also discusses how to test your application.

You can design your client application to use smaller applications to provide discrete functions. For example, a very simple bank management application could let the user choose to enter account transactions or view the current balances of the accounts. These two functions could be provided by small separate applications that are started by the bank management application.

The application that starts the other applications using the `LaunchService` service object is referred to as the *main client application*. A project in this application uses the `AppletSupport` library as one of its supplier plans.

An **applet** is a client application that is started as part of another application. In design and function, an applet is actually a small independent application. When you make the application distribution, you mark this application as an applet, whose client partition is to be started only by using the `LaunchMgr.RunApplet` from another main client application.

When you install a client application that starts applets or applications, you must also install the required applets or applications independently. Ideally, you should have a script for this installation, as describe in [“Configuring and Deploying the Main Client Application” on page 441](#).

## Using the LaunchService Service Object

The `AppletSupport` library defines a user-visible service object called `Launcher`, which is of type `LaunchMgr`. This service object provides the same services that are provided by the Launch Server (`ftlaunch`), as described in *iPlanet UDS System Management Guide*.

If you start the main client application using the Launch Server, then the `LaunchService` service object in that application directly represents the Launch Server.

If you start the application independently of the Launch Server, using the `ftexec` command or by running its compiled executable, then a `TOOL` reference to the `LaunchService` service object starts a `LaunchService` service object in the same process as the application. This `LaunchService` service object does not represent the Launch Server (`ftlauncher`) that might also be running on the client node; it is an instance of the `LaunchService` service object running in the current process.

When your application needs to access methods and events on a `LaunchMgr` object, reference the `LaunchService` service object. Do not create a new instance of the `LaunchMgr` class. The methods available for the `LaunchService` service object are described in [“Using LaunchMgr Methods” on page 439](#). For detailed descriptions of the methods and events in the `LaunchMgr` class, see Framework Library online Help.

Always assign the partition containing the user-visible `LaunchService` service object to a client node.

## Using LaunchMgr Methods

The LaunchMgr class provides the following methods, which can be invoked on the LaunchService service object:

**ListApplets** Returns a list of the available applets or applications. Use this method to see which applications and applets are loaded in the environment and available to the client node. This list can include both assigned and publicly-available applications and applets.

**RunApplet** Starts the specified applet or application, downloading the latest image repositories, if necessary.

**RunningApplets** Returns a list of the applets or applications that were started by the LaunchService service object.

**Shutdown** Shuts down one or more applets or applications that were started by the LaunchService service object, possibly including the LaunchService service object itself.

**UpdateApplet** Updates the specified assigned applet or application.

For more detailed information about these methods, and about the events available on the LaunchMgr class, see Framework Library online Help.

## A Scenario

iPlanet UDS provides an example in which a main client application can start two other applications, which have been deployed as applets.

The example used in this section is available in the FORTE\_ROOT/install/examples/frame directory, and requires the apltbank.pex, banksvc.pex, banking.pex, and bankrec.pex files. For information about setting up this example, see [“AppletBanking” on page 633](#).

This example is a very simple bank management application, which lets the user choose to enter account transactions or view the current balances of the accounts. These two functions are provided by small separate applications that are started by the bank management application.

The main client application checks whether the applets are available on the system using the `Launcher.ListApplets` command, as shown:

**Code Example 12-1** Checking applet availability using the `ListApplets` method

```
-- Check to make sure that needed applets are available.
appletList : GenericArray of AppletData;
appletList = LaunchService.ListApplets(type = APPLETTYPE_ALL,
    category = APPLETCATEGORY_APPLETS);

foundManageAcct : boolean = False;
foundViewAcct : boolean = False;

for i in appletList do
    if (i.AppletName.Value = 'Banking') then
        foundManageAcct = True;
    elseif (i.AppletName.Value = 'BankRecords') then
        foundViewAcct = True;
    end if;
end for;
```

**Project:** AppletBanking • **Class:** BankMgmt • **Method:** Display

The main client application then indicates to the user that certain functions are not available, if the applet is not available on the client machine.

You could choose to handle this kind of situation in different ways, including:

- dynamically disable the parts of the application that rely on the missing application or applet
- provide a message indicating the problem, then end the application

In this simple example, the user clicks a button to start one of the applets. In the `TOOL` code, the main client application starts the applet using the `RunApplet` method, as shown:

**Code Example 12-2** Starting an applet using the `RunApplet` method

```
when <Manage_Button>.Click do
    self.BankingRunInfo = LaunchService.RunApplet(name = 'Banking');
    self.<Manage_Button>.State = FS_INVISIBLE;
    self.<Stop_Managing_Button>.State = FS_UPDATE;
    self.window.UpdateDisplay();
```

**Project:** AppletBanking • **Class:** BankMgmt • **Method:** Display



When the user chooses to quit the main client application, the other applets are automatically shut down along with the main client application. This application also provides buttons for shutting down the launched applications. When the user clicks one of these buttons, the main client application uses the `LaunchMgr.Shutdown` method to shutdown the launched application.

**Code Example 12-3** Shutting down applets using the Shutdown method

```
-- Stop the Banking application.
when <Stop_Managing_Button>.click do
  LaunchService.Shutdown(appletID = BankingRunInfo.AppletID);
  self.<Manage_Button>.State = FS_UPDATE;
  self.<Stop_Managing_Button>.State = FS_INVISIBLE;
  self.window.UpdateDisplay();
```

**Project:** AppletBanking • **Class:** BankMgmt • **Method:** Display

The main client application also catches the `LaunchMgr.RunCompleted` event so that it can let the end user restart an applet if it has completed running.

```
when LaunchService.RunCompleted(appID = appletID) do
  if appID = BankingRunInfo.AppletID then
    self.<Manage_Button>.State = FS_UPDATE;
    self.<Stop_Managing_Button>.State = FS_INVISIBLE;
    self.window.UpdateDisplay();
  elseif (appID = BankRecordsRunInfo.AppletID) then
    self.<View_Button>.State = FS_UPDATE;
    self.<Stop_Viewing_Button>.State = FS_INVISIBLE;
    self.window.UpdateDisplay();
  end if;
```

**Project:** AppletBanking • **Class:** BankMgmt • **Method:** Display

## Configuring and Deploying the Main Client Application

Deploying a main client application that starts other applications or applets also involves deploying the other applications or applets.

When you configure an application to be started by a main client application, you can define the application as an applet, which is intended to be started only using the `LaunchMgr.RunApplet` command.

► **To configure an application as an applet**

1. Define the client partition of the application as an applet.

In the Partition Workshop, open the Logical Partition Properties dialog for the application's client partition and click the Applet toggle, as shown:

In Fscript, use the **SetAppletFlag** command, as shown in the following example:

```
SetAppletFlag 1
```

Each application or applet is deployed using a separate application distribution. To ensure that all the applications and applets required by a main client application are installed:

- include all the application distributions with the main client applications
- write an Escript script that installs all the applications and applets when the main client application is installed

The following example shows how an Escript script could install all the applets used by the main client application, if the applets are assigned:

```
LoadDistrib AppletBanking c10 # The main client application.
Install
FindActEnv                    # Back to the Environment agent.
LoadDistrib Banking c10
Install
FindActEnv
LoadDistrib BankRecords c10
Install
```

Whether the applications or applets used by the main client application are assigned or publicly available, the system manager can simply use a script, like the one in the previous example, to make the applications or applets available to the appropriate client nodes.

If you choose to start the main client application independently of the Launch Server (`ftlaunch`) application, you need to install the client partition for the main client application.

If none of the other applications or applets have server partitions, you could simply load the distributions for all the applications or applets, and let the `LaunchService` service object automatically download the applications on the clients. In fact, the only partitions you need to explicitly install are the server partitions.

If the client partition and one or more server partitions are assigned to a client node that can run server partitions, such as UNIX with Motif or Windows NT, the Launch Server downloads any assigned server partitions at the same time it installs the client partition.

If these applications or applets are publicly available, then the system manager needs to manage them the same as she would for other publicly-available applications, as described in the *iPlanet UDS System Management Guide*.

## Testing

When you test an application that launches iPlanet UDS application, be aware that any applications or applets that are being launched must be available in the development environment (installed or publicly available). However, the application that launches these client applications or applets does not need to be installed if you are testing the application in the iPlanet UDS Workshops or using the **Run** or **RunDistrib** commands in Fscript.

## Customizing the Launcher Application

iPlanet UDS provides the source code for the Launcher application as an example that you can examine and customize for your own purposes. This code is provided in the `FORTE_ROOT/install/examples/frame/launcher.pex` file.

To customize this application, import the `AppletSupport` library in the `$FORTE_ROOT/userapp/appletsu/cl0/apltsupp.pex` file, then import the code for the Launcher application. If you wish to provide help, you should create your own help file.

## Deploying Applications That Launch Other Applications And Applets

By using the `LaunchMgr.RunApplet` method, application developers can design applications that start other iPlanet UDS client applications in the same `ftexec` process, but in separate partitions.

An *applet* is a client application that is started as part of another application. In design and function, an applet is actually a small independent application. When the application developer makes the application distribution, she marks the application as an applet, whose client partition is to be started only by using the `LaunchMgr.RunApplet` from another “main” client application.

You can see whether an application distribution is an applet by checking the Partition agents for the client partition. In the Environment Console, the Type for the client partition says Partition (Applet). In Escript, the Partition Type for the Partition is Client (Applet).

Applets are deployed in exactly the same way as other applications, except that no icons are generated for applets, and applets cannot be started by the Launcher application. Applets can either be explicitly assigned to a client node, or can be publicly available to all client nodes, just as other applications can. For specific information about assigning applets or making them publicly available to client nodes, see the *iPlanet UDS System Management Guide*.

When you deploy a client application that starts applets or applications, you must also deploy the required applets or applications independently. Ideally, you should have a script for this installation.

After you have deployed an applet, you will notice that it is not visible on the client node the way a regular client application is:

- the Launcher application does not display an applet as an application that can be started
- iPlanet UDS does not automatically create icons for this applet the way it does for other applications on the Windows platforms
- the `ftcmd list` command does not include applets in its list of available applications

## Troubleshooting Client Applications That Use Applets

If you have problems with a client application that starts other applets or client partitions, you can diagnose the source of the problems by checking the trace window or log file for the main client application. The application prints trace information about the application itself and the applications or applets it starts in the same trace window or log file.

The Launch Server automatically turns on the trace flag that has the trace flags start with the name of the application: "cfg:sp:8." You can turn this trace flag on or off in other situations, if you wish.

If the main client application generates a UsageException exception with a SP\_ER\_INVALIDSTATE reason code, then one of the applications or applets it starts is missing. See the RunApplet method in Framework Library online Help for information about exceptions that can be raised when the main client application starts another application.

## Developing Applications with Nomadic Clients

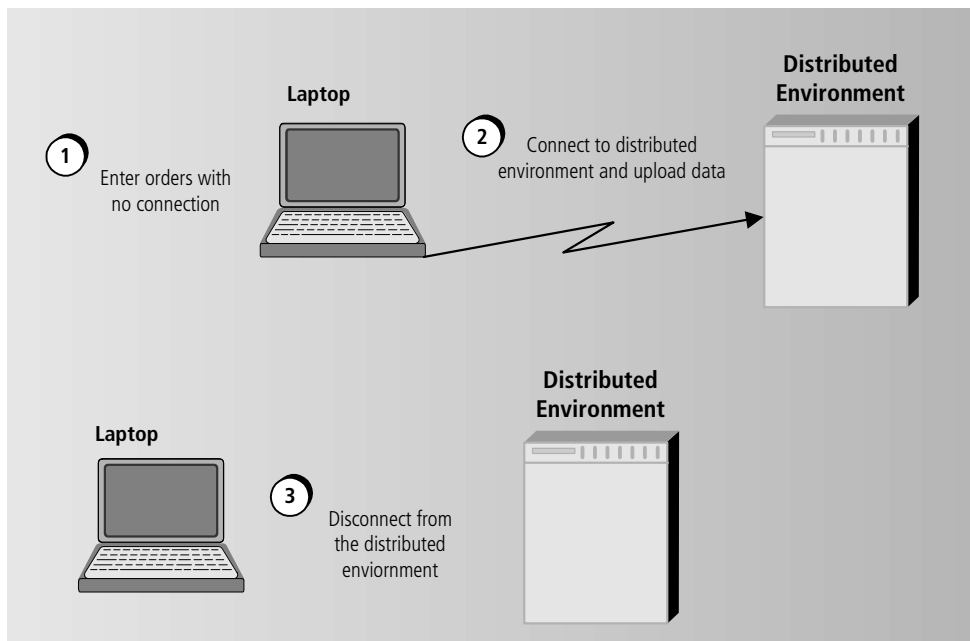
In a mobile computing environment, an application's client partitions should be able to perform many functions without being connected to a server, and should connect to a distributed environment only when they need to. This section refers to clients running in this type of application as nomadic clients. Nomadic clients are useful when the end user does not have an easily available network connection and needs to use less efficient communication devices, such as modems.

Nomadic clients can perform many functions without connecting to any environment or server. When the application needs to connect to server partitions in the environment, it does. When the client has finished using the function that requires a connection, it can disconnect from the environment and servers and run independently.

**Figure 12-1** illustrates how the NomadicOrderClient example application, which has a nomadic client partition, works. This application is described in "**NomadicOrderClient**" on page 642. In this application:

1. The client partition can run independently when the user enters the sales order information.
2. When the order information has been entered, the user can connect to her company's distributed environment, perhaps using a modem, and upload her orders to a server.
3. When the client no longer needs to access the servers in the environment, the client partition can disconnect from the environment.

**Figure 12-1** Running an Application using Nomadic Clients



When you design applications with nomadic clients, you need to consider what functions the clients must be able to perform without being connected to any server partitions. You should have the client partition access service objects and anchored objects as isolated incidents, in which the client connects to the service object, performs needed functions, then disconnects from the environment.

You can use the following features to implement nomadic clients in your application:

**Distributed reference to a service object or anchored object** Connects to an environment and the referenced service object or anchored object. For example, a distributed reference can be a method call on an environment-visible service object, such as `MyServices.GetData();`

**DistObjectMgr.ReleaseConnection method** Releases the connection to the specified service object or anchored object.

**DistObjectMgr.ReleaseNameService method** Releases the connection to the environment's Name Server.

**-fnomad flag** Starts a client application as a stand-alone application without initially connecting to an environment.

The following sections describe how to use these features to implement nomadic clients.

## Connecting to the Environment

When a client partition uses the **-fnomad** flag to start as a nomadic client, there is no initial connection to the environment. As soon as the client references a service object or anchored object, however, the client automatically connects to an environment. The client partition will stay connected to this object until you invoke the `DistObjectMgr.ReleaseConnection` method and will stay connected to the environment until you invoke the `DistObjectMgr.ReleaseNameService` method.

In the following example, `OrderServer` is an environment-visible service object. Invoking the `FindOrder` method on the `OrderServer` service object causes the client partition to connect to its default environment to locate the service object.

### Code Example 12-4 Connecting a client partition to the environment

```
-- This reference to the environment-visible service object
-- makes this client partition connect to the environment.
foundOrder = OrderServer.FindOrder(custID = self.CustomerID);
```

**Project:** NomadicOrderClient • **Class:** FindOrderWindow • **Method:** Display

To specify which environment the client should connect to, use the `FORTE_NS_ADDRESS` environment variable or the **-fns** command line flag to specify one or more name service addresses. The first name service address specifies the default environment. You can include other name service addresses so that the client can connect to another environment if its default environment is not available. The client always tries to connect to the first listed name service, then the second, and so forth.

For example, if the user usually works in California, the name service address for the default environment might be `Oakland:5000`. However, the user might occasionally take business trips to Brazil, Japan, and Spain, so other name service addresses might be specified, as shown in the following example:

```
-fns Oakland:5000;Brazil:7000;Japan:4500;Spain:3500
```

In this example, Brazil, Japan, and Spain are host names of different Environment Managers.

If the client partition disconnects from the environment, then references a service object or anchored object, iPlanet UDS automatically establishes a new connection to the environment. However, if a service object has been replicated, the client might not connect to the same instance of the service object when it reconnects.

## Disconnecting the Client Partition from the Distributed Environment

When you design your application, you should consider at what points the client partition can disconnect itself from service objects, anchored objects, and the Name Server, and plan to invoke the `DistObjectMgr.ReleaseConnection` and `DistObjectMgr.ReleaseNameService` methods at those points.

To completely disconnect the client partition from the distributed environment, you must release the client partition's connections to each service objects and anchored object, as well as the Environment Manager.

### Releasing the Connection to Service Objects and Anchored Objects

The `DistObjectMgr.ReleaseConnection` method releases a partition's connection to a service object or anchored object that the client partition has referenced.

The client partition can release a connection to a service object or anchored object at any time by using the `DistObjectMgr.ReleaseConnection` method. You can access the distributed object manager for the client using the identifier "task.Part.DistObjectMgr". The following example shows how you can disconnect from a service object named `OrderServer`:

#### **Code Example 12-5** Disconnecting from a service object

```
task.Part.DistObjectMgr.ReleaseConnection(object = OrderServer);
```

**Project:** NomadicOrderClient • **Class:** FindOrderWindow • **Method:** Display



If the client partition has accessed several service objects and anchored objects, you need to invoke the `ReleaseConnection` method for each environment-visible service object and anchored object. You then need to invoke the `ReleaseNameService` method, described in the next section, to completely disconnect the client partition from the environment.

---

**NOTE** The `DistObjectMgr.ReleaseDistReference` method releases the reference to an anchored object or service object, but does not release the connection to the partition containing the object.

---

## Releasing the Connection to the Environment

The `DistObjectMgr.ReleaseNameService` method releases a partition's connection to the Environment Manager called.

The client partition can release a connection to the environment at any time by using the `DistObjectMgr.ReleaseNameService` method. You can access the distributed object manager for the client using the identifier "task.Part.DistObjectMgr". The following example shows how you can disconnect from the current Environment Manager:

### Code Example 12-6 Disconnecting from an environment

```
task.Part.DistObjectMgr.ReleaseNameService();
```

**Project:** NomadicOrderClient • **Class:** FindOrderWindow • **Method:** Display

## Example of Connecting and Disconnecting to the Environment Manager

The following sample code from the `NomadicOrderClient` application shows how a client partition can connect to the Environment Manager and a service object named `OrderServer` by calling a method on the service object. The client partition then disconnects from the service object and Environment Manager when it has completed its task.

### Code Example 12-7 Connecting and disconnecting a client partition

```
-- This reference to the environment-visible service object
-- makes this client partition connect to the environment.
OrderServer.AddOrder(newOrder = index);
.
.
-- Use the ReleaseConnection method to disconnect from
-- the service object.
task.Part.DistObjectMgr.ReleaseConnection(object = OrderServer);
task.Part.LogMgr.PutLine(source=
  'Disconnected from the remote service object OrderServer.');
```

**Project:** `NomadicOrderClient` • **Class:** `StartWindow` • **Method:** `Display`

## Starting a Nomadic Client Application

If a client does not need to immediately access distributed objects, you can specify the **-fnomad** flag as part of the client's startup command. When this flag is specified, the client does not connect to an environment when it starts up.

You can use the **-fnomad** flag with any command that starts a client partition, including the following:

- `ftexec` (portable syntax)
- `VFORTE FTEXEC` (VMS)
- **ftclient** or **ftclntds**
- compiled partition executables

You should not run nomadic client applications with the **ftcmd run** command, because the `DistObjectMgr.ReleaseNameService` method causes the Launch Server, as well as the client partition, to disconnect from the environment.

The default icons created for client applications use the **ftcmd run** command to start the application. To have iPlanet UDS generate icons that start applications using the **ftexec** command, set the following configuration flag on each client node where you want this type of icon generated: `cfg:em:2`. Alternatively, you can edit the icons to use the **ftexec** command.

The following example shows how to specify that the client partition starts without connecting to an Environment Manager. This example shows the syntax for a client application on Windows NT:

```
ftexec -fi bt:c\forte\userapp\ordercli\cl0\orderc0 -fnomad
```

In Windows NT, this command could be entered on the Command Line field of the Program Item Properties dialog for the program icon that represents the OrderClient application.

## Testing Nomadic Client Applications

When you test run your application using the iPlanet UDS Workshops or the Fscript utility in distributed mode, you need to comment out any lines that contain the `DistObjectMgr.ReleaseNameService` method. Otherwise, when the client disconnects from the Environment Manager, Fscript or the iPlanet UDS Workshops also lose the connection to the environment, which will cause Fscript or the iPlanet UDS Workshops to fail. The only functional difference between including the `ReleaseNameService` method call and commenting it out should be that the client partition stays connected to the environment after it initially connects.

You can only test that your client partition properly disconnects and reconnects to the Environment Manager by partitioning, making a distribution, and installing the application in a test environment. You then need to start the client partition using the **-fnomad** flag, as described in [“Starting a Nomadic Client Application” on page 450](#).

Because you can only test your application with nomadic clients by deploying and running the application, design your application to print a lot of messages to a log file or trace window. You can then use these messages when you run your application to determine when failures occur and what causes them. You should also design useful exceptions into your application, so that your application can intelligently recover from error situations.

## Restrictions on Nomadic Clients

Typically, when a client partition connects to an environment during its startup, the client partition becomes part of that environment, and that environment's Environment Manager is the client partition's home Environment Manager.

If the client does not initially connect to an Environment Manager, the client forms its own environment without an Environment Manager and never has a home Environment Manager. Therefore, if the client starts with the **-fnomad** flag, you need to design the client with the following restrictions in mind:

- The client cannot register any objects using the `ObjectLocationMgr` class, because this feature does not work if the client does not have a home Environment Manager.
- If the client has for some reason failed over to a different Environment Manager, the client does not disconnect from the second Environment Manager and reconnect to the first Environment Manager it connected to, the way clients do when they have a home Environment Manager.

# Upgrading Deployed Applications

This chapter describes how to upgrade iPlanet UDS user applications that are currently deployed. It covers advanced topics of interest to iPlanet UDS system managers as well as application developers, including:

- when to use interoperable upgrades, compatibility level upgrades, and class version upgrades
- how you perform each type of upgrade
- the use of compatibility levels and class versions
- how you write converters to modify a deployed application
- how you track changes in releases of an application

Note that this chapter describes upgrading *deployed applications* only. For information regarding upgrading iPlanet UDS, refer to the *iPlanet UDS System Installation Guide*.

## Choosing an Upgrade Approach

Maintaining an application requires changing it. Changes are necessary for a variety of reasons, such as to add functionality, to reflect new business conditions, to improve performance of an application, to fix bugs, or to take advantage of new features or a new understanding of existing features. Deployed iPlanet UDS applications require updating for the same reasons. As business requirements change over time, you will very likely need to refine the class definitions and the logic of your application.

The upgrade process can be significant. To upgrade an iPlanet UDS production environment requires careful planning to execute a sequence of steps. You must make the desired changes, repartition, and install newer partitions, perhaps on a significant number of client nodes. You typically want to plan changes to an application carefully, to group related changes when possible, and to minimize downtime, retraining, and inconveniences to the end users. You may also need to coordinate the activities and needs of application developers, system managers, and end users.

When you upgrade an application you typically have the following goals:

- to make the desired modifications to an application
- to upgrade all servers and users of the application
- to do so with the least interruption and inconvenience to users, developers, and managers of the application

Careful planning and testing are critical to meet these goals as you upgrade deployed applications. Also, you should be sure to test each step of the process thoroughly.

## Types of Upgrades

When choosing an upgrade approach you must take into consideration a number of factors. Depending on your circumstances, you may be able to perform a relatively straightforward upgrade, or you may need to plan several phases to accomplish a more complex upgrade.

This chapter describes three upgrade approaches to meet a variety of requirements. These approaches are based upon characteristics of iPlanet UDS applications that affect upgrading. These approaches are listed in order of the degree of change permitted.

**Interoperable Upgrades** In an interoperable upgrade, you can make limited changes to an application, as long as you do not impact the interaction between the application's clients and servers. You upgrade the application by replacing some partitions with a newer application distribution; the changes in the newer partitions transparently work with the older partitions.

**Class Version Upgrades** In a class version upgrade you can make significant changes to an application, such that clients and servers can be based upon multiple distributions of the same application during the upgrade process. You do not (cannot) increment the compatibility level. However, you use *class versions* to

identify classes that have changed and you write special methods called **converters**. This upgrade approach is recommended primarily for applications that must always be running (“high-availability” or 7 x 24 applications) or applications where the time to distribute software to all parties may be very long (weeks).

**Compatibility Level Upgrades** In a compatibility level upgrade you replace an older application with a newer application and increment the iPlanet UDS compatibility level. The older and newer applications are completely independent. Usually you upgrade all partitions simultaneously; however, under some circumstances you can upgrade partitions in groups or phases.

These upgrade approaches are also summarized in the following table.

	<b>Interoperable Upgrade</b>	<b>Class Version Upgrade</b>	<b>Compatibility Level Upgrade</b>
<b>Changes Allowed</b>	Add new classes, attributes, or events.	Add any new class or class component. Change method signatures and replace events (add parameters).	All changes. Only approach that allows you to delete class components.
<b>Characteristics</b>	No change in compatibility level. Failover is optional.	No change in compatibility level. Requires failover. Uses class versions and converters.	Compatibility level must change. Cannot use failover.
<b>Advantages</b>	Allows a rolling upgrade.	Allows a rolling upgrade. Allows many phases or versions of an application, if necessary.	Requires no special coding. Under some circumstances allows a rolling upgrade.
<b>Disadvantages</b>	Most restrictive on types of changes allowed.	Requires coding for converters. More complex to administer.	Difficult or impossible to schedule for high-availability applications. A rolling upgrade is not always possible.

Each upgrade approach has different assumptions and steps. The first half of this chapter describes some of the reasons that you would choose one approach over another. The second half of this chapter describes how to perform these upgrades.

## Factors Influencing Upgrade Possibilities

The requirements of your business will dictate the modifications that you need to make to your application, but business conditions may also impose some restrictions on how and when you can make the modifications. Along with prioritizing the changes you must make, you should also consider the following factors as you plan your upgrade.

### When is a Rolling Upgrade Necessary?

Rolling upgrades are a characteristic of distributed applications only. One partition of an iPlanet UDS application can contain only one release of an application. However, a typical iPlanet UDS application is distributed across many partitions, sometimes making it difficult to upgrade all of the partitions simultaneously. Furthermore, if an application must guarantee around-the-clock availability, compatibility level upgrades are rarely possible. For highly distributed or high availability applications, the ability to perform rolling upgrades is critical.

Some typical situations that require rolling upgrades are described below.

**Applications that are geographically or logically dispersed** Client partitions may be installed on many workstations distributed over a wide geographical area. Or, local iPlanet UDS system managers may be responsible for upgrading regions or types of clients, subject to their own scheduling constraints. This situation could result in clients being upgraded at different times, or a temporary mix of client releases.

**Applications that require high availability** High-availability applications must reliably be available for use at all times. Upgrades must be performed while the application is running, with no detrimental impact on current users of the application. An upgrade of servers for a high-availability application may require a temporary mix of releases.

Upgrades in high-availability applications typically require careful planning to define the exact upgrade procedure and to allow for backup if something should go wrong. In addition, there is usually a window of time during which multiple versions of the software must work together (a subset of older software that is being replaced must work with a subset of newer software that is replacing it). This window of time can be significant; it can extend from a few minutes to several months, depending upon what is delaying the upgrade of the remaining older components.



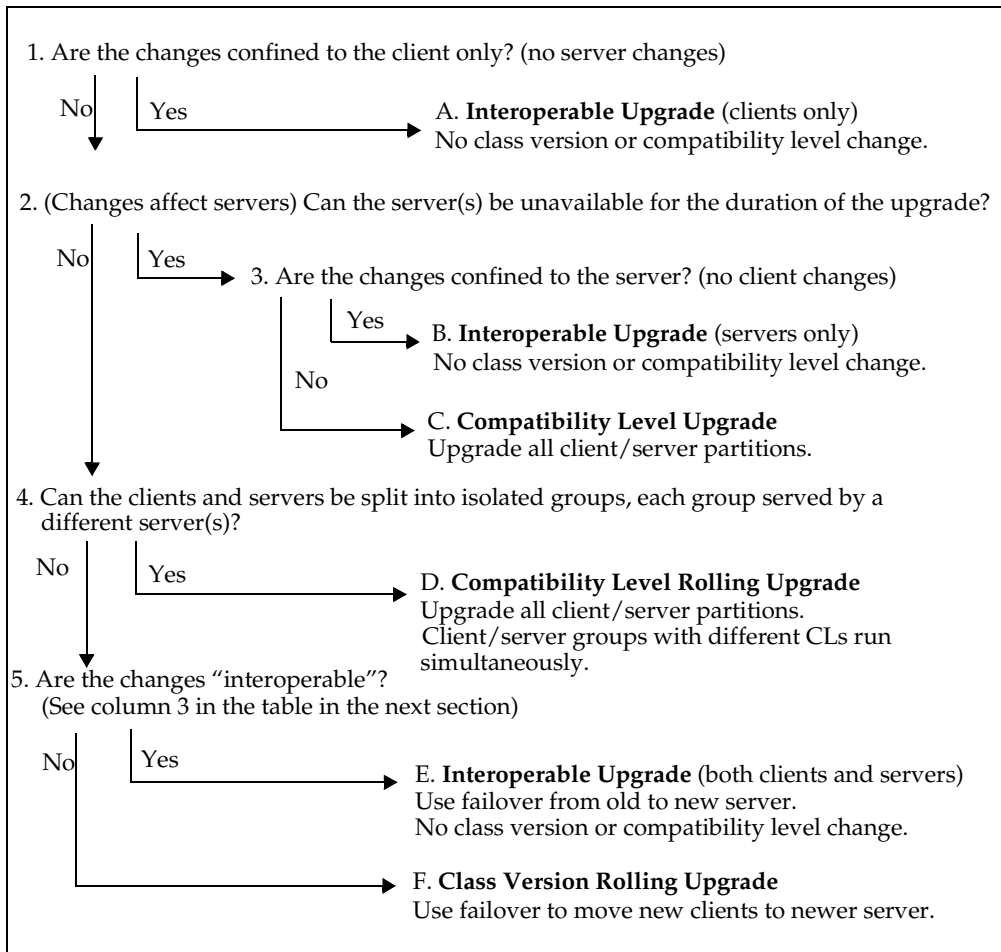
**Applications in which all clients must access the same server** In many applications, all clients must access the same data and have immediate access to updates made by other clients. In these situations, there cannot be two separate servers (which can simplify an upgrade); instead there must be either one central server that can be accessed by all clients, or a replicated server, in which multiple servers stay synchronized.

**Applications of different levels that share resources** You might have multiple iPlanet UDS applications that share common services or data. If it is impossible or unreasonable to make simultaneous updates to all clients of all applications, you can still phase in updates to the applications over time. While iPlanet UDS applications that share resources must have the same compatibility level, it is possible to change class definitions within a compatibility level in order to add new functionality without requiring a complete upgrade.

Some rolling upgrades require the use of iPlanet UDS's failover feature when upgrading servers. An interoperable or class version rolling upgrade requires failover, so that clients can move immediately and transparently from an older server to a newer one. Note that to use failover, all partitions must be at the same compatibility level.

By answering the questions in [Figure 13-1](#) you can determine which type of upgrade you need to make.

**Figure 13-1** Upgrade Approach Decision Tree



Each situation imposes different requirements on you as an iPlanet UDS application developer to guarantee that the application is consistent and that changes are readily incorporated into the application. You may have to trade off or delay some changes due to your particular circumstances.

## Changes Allowed Between Upgrades

The changes that you can make to an application depend upon the upgrade approach.

You can make *any change* to an application during a compatibility level upgrade. You are essentially creating a new application. Some changes, particularly deleting class components, are restricted to this type of upgrade.

You can make certain types of changes to an application during interoperable and class version upgrades, as shown in the table below. If there is a “no” in both columns 3 and 4, then the change can only be achieved using a compatibility level upgrade. Also see below for an important note regarding the table.

Component	Allowable Changes	Interoperable Upgrade	Class Version Upgrade
<b>Classes</b>	Add a new class with the distributed property	yes (See Note 1)	yes (See Note 1)
	Add a new class that is non-distributed but is sent (as a parameter or an attribute) to an older version	no (See Note 2)	no (See Note 2)
	Delete a class	no	no
<b>Interfaces</b>	No effect. (Interfaces are not distributed. Classes that implement interfaces can be distributed.)	no effect	no effect
<b>Service Objects</b>	Add a new service object	yes (See Note 1)	yes (See Note 1)
	Delete a service object	yes (See Note 1)	yes (See Note 1)
<b>Attributes</b>	Add a new attribute	yes	yes
	Delete an attribute	no	no
<b>Methods</b>	Add a new method	no	yes
	Change a method’s signature or materially change a method’s behavior.	no (See Note 4)	yes (See Note 3)
	Delete a method	no	yes
<b>Events</b>	Add a new event	yes (See Note 1)	yes (See Note 1)
	Replace an event (that is, change event parameters)	no	yes (See Note 3)
	Delete an event	no	no
<b>Event Handlers</b>	Add a new event handler	no effect	no effect
	Delete an event handler	no effect	no effect

Component	Allowable Changes	Interoperable Upgrade	Class Version Upgrade
NOTES			
<ol style="list-style-type: none"> <li>1. Permitted as long as an instance of the service object/class/event is available to all partitions that need it.</li> <li>2. A non-distributed class cannot be sent to an older partition that does not recognize it.</li> <li>3. When a parameter associated with a method or event is changed, it is treated as the creation of a new method or event.</li> <li>4. Materially changing a method's behavior means that the change is such that a client expecting the former behavior will be disappointed.</li> </ol>			

When performing a rolling upgrade on a distributed application, you must guarantee that older components interoperate with newer components. However, you need only be concerned with components that are *distributed*. (If a component is local to one partition, it can never be incompatible with itself.) Therefore this chapter (and the column “Allowable Changes”) describes changes that you can make to classes in either of the following categories:

- classes that have the distributed property
- classes that may not be distributed but are passed between partitions as parameters to method calls

If a class does not have the distributed property and is never passed between partitions, you can freely change its definition.

## About Class Versions and Compatibility Levels

iPlanet UDS uses the concepts of compatibility levels and class versions to identify unique snapshots of iPlanet UDS application components, as described below.

A *compatibility level* is an integer assigned to an application to identify components that can work together. Multiple releases of an application can co-exist on a node without interference, as long as they are identified by different compatibility levels. Without compatibility levels, two releases of the same application could “collide” on a node.

Compatibility levels are integer values; the actual compatibility level for an application and all its components is set when an application is partitioned (see *A Guide to the iPlanet UDS Workshops*). The compatibility level for an application is taken from the compatibility level for the main project of the application. The compatibility level is appended to the string “CL,” as in CL0.

For example, a node can run two server partitions (one CL1 and the other CL2) for the same application. Clients running CL1 would automatically communicate with the CL1 server partition, and CL2 clients would automatically connect with the CL2 server partition; both “applications” could run in parallel, even though the application components use the same names. This is possible because the compatibility level is added to each partition’s name, making each name unique.

A library has a compatibility level based on the compatibility level of the project for which you used the Configure As Library command. Library compatibility levels are used just like application compatibility levels. If your application references a library, then you must update your application if the library is updated, its compatibility level changes, *and* you want your application to take advantage of the newer library. See [“Using New Compatibility Levels of Libraries and Shared Service Objects” on page 479](#).

For information about using compatibility levels, including when to increment them, see [“Using Compatibility Levels to Upgrade” on page 478](#).

A *class version* is a runtime property of a class that uniquely identifies one definition of that class (all its attributes, methods, events, and so on). Within a compatibility level a class can have multiple versions. While only one version of a class can be loaded in a single partition, a deployed application can have multiple versions of one or more classes (on different partitions) that work together. For more information about using class versions, see [“Using Class Version Numbers” on page 492](#).

Class versions require hand-coded *converters* (a special type of method) so that partitions based on different class versions can communicate during the upgrade process. Converters contain code that makes up for differences between the class versions. For more information about converters, see [“About Converters” on page 467](#).

## About Interoperable Upgrades

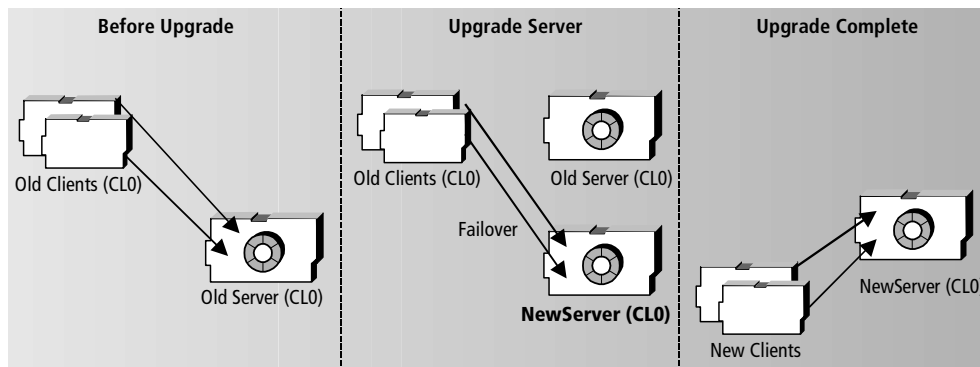
In an interoperable upgrade you can modify an application without raising its compatibility level if you follow certain rules to guarantee runtime interoperability (sometimes called “compatibility”). *Interoperability* refers to the ability of partitions from different distributions of the same application to communicate successfully, without the use of class versions or converters.

The changes you make to the application must not introduce any incompatibilities between older and newer partitions. If you need to make changes that will change the interaction between clients and servers, then you must upgrade using one of the other two approaches.

In an interoperable upgrade, you modify the application, make a new application distribution, and replace some old partitions with new partitions. You may update either client or server partitions only, or a selective mix of both.

In [Figure 13-2](#) selected partitions are simply updated with a new distribution. Changes made between the older and newer distribution are transparent to iPlanet UDS application users (the changes do not impact any interaction between clients and servers). Note also that failover is used, so that service to the clients is not interrupted.

**Figure 13-2** Interoperable Upgrade with Failover



### Upgrading Clients

An interoperable upgrade to clients is possible only when you need to change just the client behavior in your application; you require no changes to the server. For example, you might simply rearrange the display of data on client screens.

You can upgrade all client partitions, or only a portion of them. As you upgrade each client you need only interrupt access for that client; the upgrade has no effect on the server or other clients.

## Upgrading Servers

Many upgrades require changes to server partitions. Server upgrades frequently require corresponding changes in clients, but this is not always true.

If the server can be unavailable to clients during the upgrade, you can simply stop your servers at a low-use time, perform the upgrade on the servers, test, and restart the servers and clients when you are satisfied that the upgrade is functional.

If the server must be available to clients during the upgrade, then you will use the iPlanet UDS failover feature (shown in [Figure 13-2](#)). However, if the server need not be available at all times, an interoperable upgrade can be performed by briefly interrupting service to the clients while the newer server partitions replace the older server partitions.

The iPlanet UDS example programs `TimeItV1-4` illustrate runtime compatibility between different distributions. See [Appendix A, “iPlanet UDS Example Applications”](#) for information about running these programs.

## About Compatibility Level Upgrades

In a *compatibility level upgrade*, you replace an entire application with a completely new release of the application. All servers and users move to a new release of an application, whether simultaneously or in phases.

A compatibility level upgrade allows any change to an application. However, all clients must be upgraded if they must connect to the same (new) server, because client partitions at one compatibility level cannot connect to servers of another compatibility level.

You cannot use failover as an upgrade technique for this type of upgrade. Failover does not work between partitions of different compatibility levels.

Some business circumstances *allow* you to perform a compatibility level upgrade to an iPlanet UDS application. And some technical circumstances *require* you to perform a compatibility level upgrade.

You can perform a compatibility level upgrade if either of the following conditions is true:

- You can upgrade all of your end users and servers simultaneously (as in [Figure 13-3](#)).

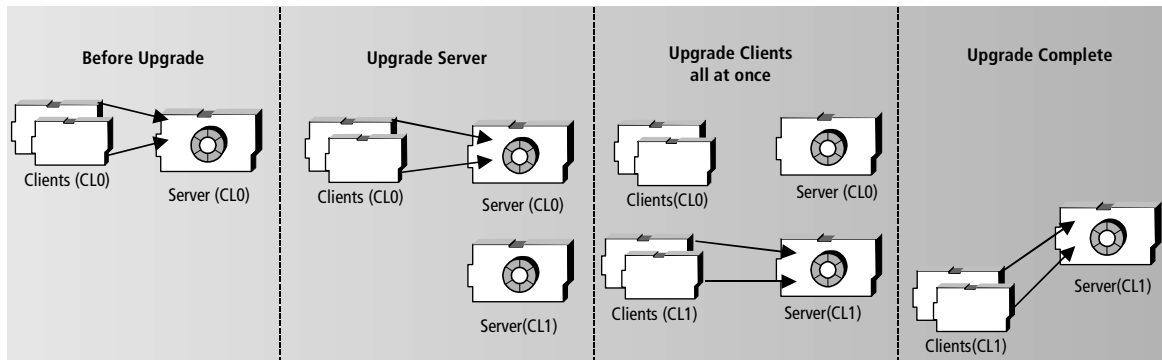
- You can upgrade a distinct subset of your end users and servers simultaneously and the nature of your application allows clients to access (and manipulate) data through different servers. In this scenario, you will have clients and servers from multiple compatibility levels accessing the same underlying data (as in [Figure 13-4](#)). This may be acceptable for some applications.

While you can make *any* change to an application during a compatibility level upgrade, there are some changes that you must not make *unless* you perform a compatibility level upgrade. These changes are:

- deletion of a distributed class, a component of a distributed class
- addition of a class component without the creation of a converter

[Figure 13-3](#) shows a simple view of a compatibility level upgrade. One benefit of incrementing the compatibility level is that you can easily distinguish newer partitions from older partitions, because the partition names include the compatibility level.

**Figure 13-3** Compatibility Level Step Upgrade

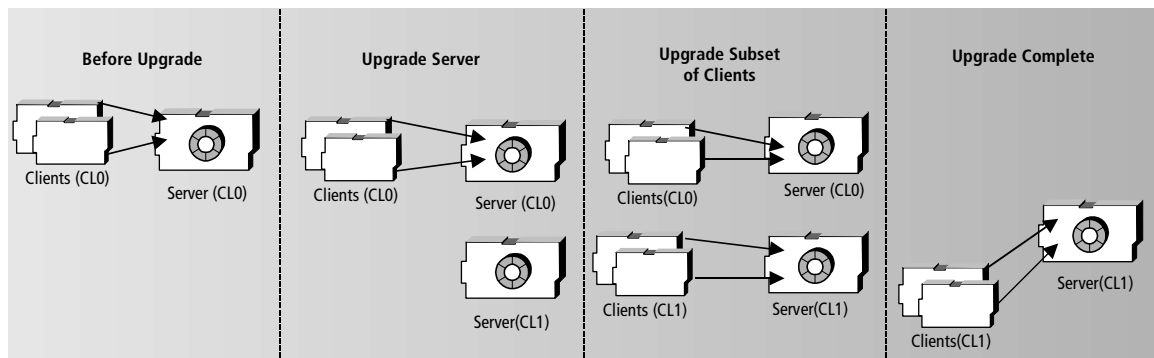


### About Compatibility Level Rolling Upgrades

If you can separate the clients and servers into isolated groups, then you may be able to perform a type of rolling upgrade with a new compatibility level.

As shown in [Figure 13-4](#), you simply replace a subset of the older application partitions, whether client or server partitions, with the newer partitions, and restart the application.



**Figure 13-4** Comptability Level Rolling Upgrade

In this case, all partitions are running the same application, but in effect, during the “Concurrent Operation” phase, two different releases are running as if they were two independent applications.

## About Class Version Upgrades

In a *class version upgrade*, old and new application partitions coexist for a period of time, until the older partitions are completely replaced by newer ones. Class version upgrades are only required for applications that can never be unavailable, to assure that a client can always reach a server.

Class version upgrades allow you to perform a rolling upgrade without having to change the application compatibility level and upgrade all clients and servers simultaneously. They also allow you to make substantial changes to class components in your application. However, you must write special methods called *converters* and distinguish older classes from newer ones using *class versions*.

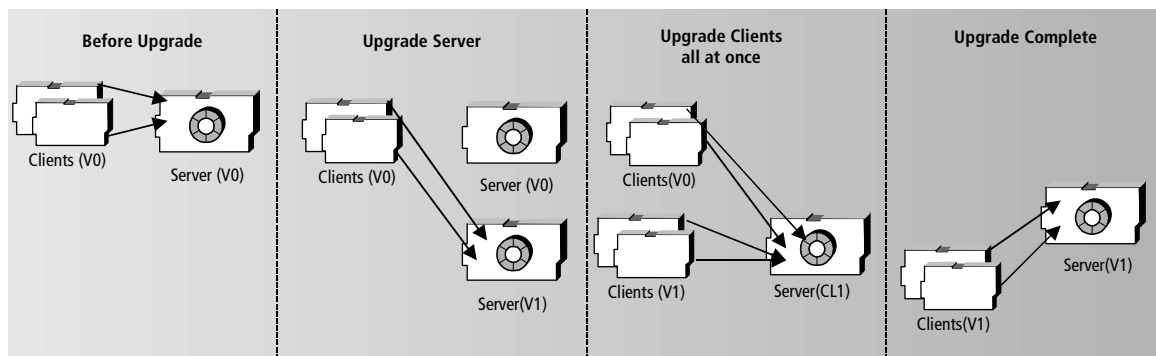
---

**NOTE** Class version upgrades are the most technically complex type of upgrade. Before you attempt a class version upgrade you should have a thorough understanding of your application, of using distributed objects, and of the iPlanet UDS system.

---

Figure 13-5 shows a simple class version level upgrade, with only one server partition. Here the server is upgraded first; however, in an application with many servers and clients, partitions could be upgraded in any order. During the “Upgrade Server” phase, the newer server partition is started; then the older server partition is stopped and the clients failover to the newer server. In the “Upgrade Clients” phase, client partitions based on different class versions all access the same, newer server.

Figure 13-5 Class Version Upgrade



The advantage of a class version upgrade is that both of the Upgrade phases can take as long as necessary (and overlap) because they do not require any interruption to the application’s availability.

Because you must guarantee availability of the application to clients, you must use failover when upgrading servers. Failover guarantees that one server partition replicate is always available to respond to clients. Because replicated partitions must be at the same compatibility level, all changes related to the upgrade must be introduced using class versions rather than compatibility levels.

Figure 13-5 does not show one aspect of a class version upgrade: the use of converters. The next section, “About Converters” on page 467 shows a simple upgrade along with the converters that are used.

The iPlanet UDS sample program Banking1-2 demonstrates the use of converters that might be used in a class version upgrade. For information about running Banking1-2, refer to Appendix A, “iPlanet UDS Example Applications.”

A class version upgrade allows to you to modify various class components, as shown in the following table.

**Table 13-1** Changes Allowed During a Class Version Upgrade

Component	Allowable Changes	Converter Required
<b>Classes</b>	Add a new class with the distributed property.	New method converters, one for each method New event converters, if new class registers for old event
<b>Methods</b>	Add a new method.	New method converter
	Change a method's signature.	New method converter and obsolete method converter
	<i>Materially</i> change a method's behavior.	New method converter and obsolete method converter.
	Delete a method.	Obsolete method converter
<b>Events</b>	Add a "brand new" event.	None, IF all partitions which access and implement the event are upgraded simultaneously
	"Replace" an event. (If you change an event's parameters, you essentially create a new event, but you do not delete the existing one.)	New event converter and obsolete event converter

For more information on how converters work, see ["About Converters"](#) below. For information about writing converters, see ["Guidelines for Writing Converters"](#) on [page 485](#).

## About Converters

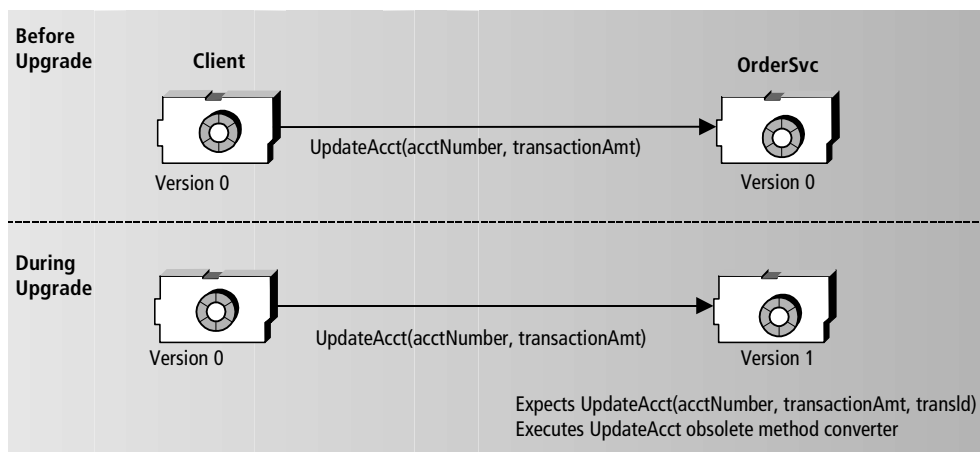
A class version upgrade requires that you write converters. A *converter* is a special method that you write to adjust for the different versions of a class component. A converter allows older and newer code to communicate by "bridging" the differences between two versions of a method or an event. Converters are used in rolling upgrades, to enable code that calls ("expects") one version of a class component (by invoking a method or posting an event) to actually use a different version of the component.

Figure 13-6 illustrates the use of an obsolete method converter. An obsolete method converter is used when the caller is older than the callee, and the caller invokes a method that no longer exists in the callee. This could occur when a server is updated before a client, as in this example: the UpdateAcct method is changed on OrderSvc to add a third parameter, transId.

The top of Figure 13-6 shows before the upgrade begins: both the client and OrderSvc use the UpdateAcct method with only two parameters, acctNumber and transactionAmt.

In the bottom of Figure 13-6 OrderSvc is upgraded and now expects UpdateAcct to be invoked with three parameters. Yet, until the client is upgraded, it can continue to use the older version of the UpdateAcct method.

Figure 13-6 Interoperable Upgrade with Failover



This is true because OrderSvc has an obsolete method converter for UpdateAcct, specifically so that it can work with older clients. So, when the older client invokes UpdateAcct with two parameters, OrderSvc automatically invokes the obsolete method converter, which simply uses a default value for the third parameter, as shown in the following line from the converter code:

```
return self.UpdateAcct(acctNumber, transactionAmt, 0);
```

Setting reasonable default values for new parameters in enhanced methods is a typical way to code obsolete method converters. In the opposite case, where a newer caller invokes a newer method on an older server, a new method converter might strip away parameters that are only used for the newer version, so that the older server can execute the method.

Converters have the following benefits:

Benefit	Explanation
Converters are transparent to end users	Converters are automatically invoked when different versions of classes exist within an environment and remote objects communicate using different versions of a method or event.
Converters allow multiple versions of a class to communicate.	By using converters, an environment can contain multiple versions of a class, yet objects of any version can communicate.
Converters isolate changes associated with one version of a class.	Because all code for a given class version is encapsulated within converters, it is easy to remove all the code for a given class version when that version becomes obsolete. You can easily remove code for older versions.

Converters have the following characteristics:

Characteristic	Explanation
Converters are only necessary for distributed objects.	Converters are required only to handle distributed references to objects that are distributed and are different versions. While a given partition can have only one version of a class, if an object is distributed, then another partition with a distributed reference to the object may expect a different version of that object.
Only methods and events can have converters.	You can not write converters on attributes or event handlers. Converters are unnecessary for event handlers because event handlers have no distributed access.
Converters run and reside in the newer partition.	Converters always reside and are executed in the partition with the newer version of the class, because converters are only necessary (and can only be added) when a newer version is added to an existing environment.

A method or event can have one or two types of converters: new and obsolete.

A *new converter* adjusts for a component that has been added to (is new in) the newer class version. A new converter handles a reference to a new component when that component does not exist in a previous version of the class, on the (older) callee. A new converter always executes on the (newer) caller.

An *obsolete converter* adjusts for a component whose signature has changed, or that has been deleted, in the newer class version. An obsolete converter handles a reference from a previous version to a component that is obsolete in the current version of the class, on the (newer) callee. An obsolete converter always executes on the (newer) callee.

The following table summarizes the differences between the two types of converters:

Type of Converter	Newer Class has	Caller is in	Callee is in
new converter	added a component	newer class	older class
obsolete converter	deleted or changed a component	older class	newer class

Which converter is required depends upon the changes you make in the newer class version (see [“About Class Version Upgrades” on page 465](#)). Some changes require two converters. To see examples of new and obsolete converters, refer to [“About Converters” on page 467](#).

The following sections provide more detail on method converters and event converters. For information on writing converters, refer to [“Updating Classes and Writing Converters” on page 483](#).

## New Method Converters

New method converters are required when you add a new method. A *new method converter* is used when a newer caller (newer class version) attempts to invoke a new method on an older callee (older class version), in which the method does not exist.

## Obsolete Method Converters

Obsolete method converters are required when you delete a method, or change a method’s signature. An *obsolete method converter* is used when an older caller attempts to invoke a method on a newer callee, in which the method does not exist (has been obsoleted).

The obsolete method converter can perform the equivalent of the obsolete method, or perhaps invoke a newer method that essentially replaces the older method.

## Converters for Modified Methods

You must create both a new and an obsolete method converter if you change a method in either of the following ways:

- if you change the method signature (if you change, add, or delete a return value, or change or add any parameter)
- if you *materially* change the method code, and thereby the effect of executing the method, such that a caller of the older version of the method would be disappointed by the results of the new version

If you modify a method, you should create an obsolete method converter for when older callers invoke the older version of the method against newer callees. Similarly, you should create a new method converter to handle newer callers invoking the newer method on an older callee. In either case, the appropriate converter is automatically invoked.

## Event Converters

Event converters allow you to “replace an event” (that is, modify an event by changing its parameters). However, because events, unlike methods, cannot be overloaded, you cannot create a second event with the same name (nor should you delete the older event). Instead, you must create a newer event, with a different name and the new parameters. Then you define both a new event converter and an obsolete event converter. Essentially, you are adding a new event and obsoleting an old event.

New event converters take the parameters of the new event and return an instance of the old event.

Obsolete event converters take the parameters of the old event and return an instance of the new event.

Event converters allow iPlanet UDS to assure that events are posted and registered for as expected, regardless of the class version for the poster or registrant. Thus, iPlanet UDS assures that both of the following are done automatically:

- Whenever a newer event is posted, the older event is posted, so that older registrants work correctly.
- Whenever a newer event is registered for, the older event is registered for, so that older posters work correctly.

Both types of event converters are associated with the class on which the event is defined, regardless of the object that posts or registers the event.

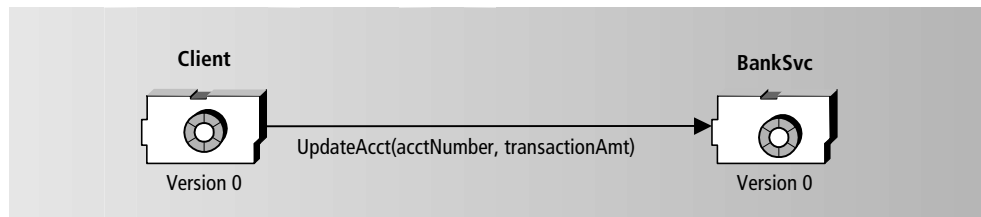
Completion and exception events do not require event converters since they are posted automatically by the system in response to a method invocation.

## The Banking1-2 Example

The iPlanet UDS sample program Banking1-2 demonstrates a very simple class version upgrade. This section describes the changes that are incorporated in the Banking application, and shows the converters that are used during the upgrade. (The Banking application is a generic distributed iPlanet UDS application upon which several other iPlanet UDS sample applications are based.)

Before the upgrade, a typical client and server interaction in the Banking application is shown in [Figure 13-7](#). Assume for the purposes of this example, however, that there are many clients, enough so that it would be difficult to upgrade all of them simultaneously.

**Figure 13-7** Before Upgrade



In this example, the client uses a window to make deposits and withdrawals from various accounts. Before making changes, all that is required for a single transaction is the account number and an amount. A negative amount is a withdrawal and reduces the balance. The BankSvc server performs the actual update of the account and responds with the new balance for the given account.

For the purposes of this example, we will make only one change. We will change the method signature for the UpdateAcct method to add a third parameter, transId. The UpdateAcct method is defined in the class BankService. This change requires us to write both a new method converter and an obsolete method converter, since we want both the following to be possible:

- new clients to access old servers (requiring the new method converter)
- old clients to access new servers (requiring the obsolete method converter)



► **To update the Banking application**

1. Change the version number for the BankService class to 1 in the Class Workshop.
2. Create the obsolete method converter for the UpdateAcct method.

---

**CAUTION** You must create obsolete method converters before changing or deleting methods. You cannot create a valid obsolete converter after having done changed a method signature or deleted the method.

You can add the code for this converter later, but you must create the obsolete converter at this point in the sequence (before updating the method).

---

3. Edit the UpdateAcct method to add the new parameter using the Method Workshop.
4. Create and code the new method converter for the UpdateAcct method.

This completes the changes we make to the BankSvc server. Now we update any method in which the UpdateAcct method is invoked.

5. Update the method Display in the TransactionWindow class to use the third parameter, transId.

The obsolete method converter is used in this example when an old client invokes the older version of UpdateAcct on a newer server. The code for the new method converter for UpdateAcct follows:

```
return self.UpdateAcct(acctNumber, transactionAmt, 0);
```

In this case the client invokes UpdateAcct using two parameters. The converter simply appends a reasonable value for the third parameter and invokes the newer version of the UpdateAcct method. This is a typical use of an obsolete converter.

The new method converter is used in this example when a new client invokes the newer version of UpdateAcct on an older server. The code for the new method converter for UpdateAcct follows:

```
return self.UpdateAcct(acctNumber, transactionAmt);
```

While the client has invoked UpdateAcct using three parameters, the converter simply strips off the third parameter and invokes the deleted version of the UpdateAcct method. This is a typical use of a new converter.

Having written the converters, we can now upgrade the client and servers.

► **To upgrade the Banking application**

1. Distribute new client software to the clients. During this step the new converter is used because some clients are newer than the server.
2. Upgrade the server. At this point, the obsolete converter is used only when older clients contact the server; clients that have been upgraded are “in sync” with the server.

Now we show two different scenarios that might occur during a rolling upgrade of this application. Neither scenario represents a problem because we have written the converters that are used automatically in either of the following cases:

- the client is upgraded and the server is not
- the server is upgraded and the client is not

Ultimately, of course, all servers and all clients are upgraded, at which point the converters are no longer used. However, until all newer partitions are installed, there will be a mix of old and new ones. In fact, both scenarios below might occur.

Figure 13-8 shows what happens if a new client should invoke UpdateAcct on server that has not been upgraded. The client invokes a method (UpdateAcct with three parameters) that does not exist on the server. In this case, the *client* automatically executes the new method converter for that method, since the new method converter exists only on the client.

**Figure 13-8** Caller is Newer: New Converter Executes Caller

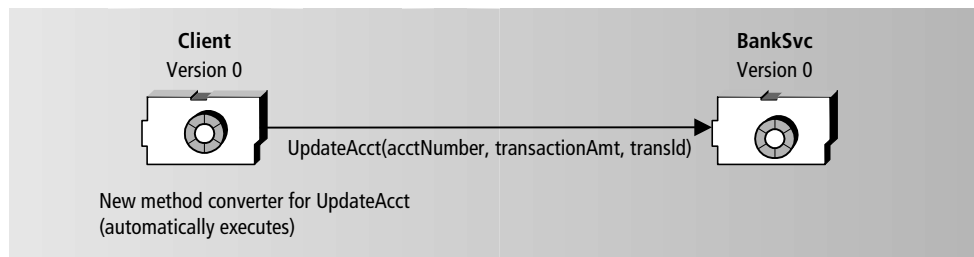
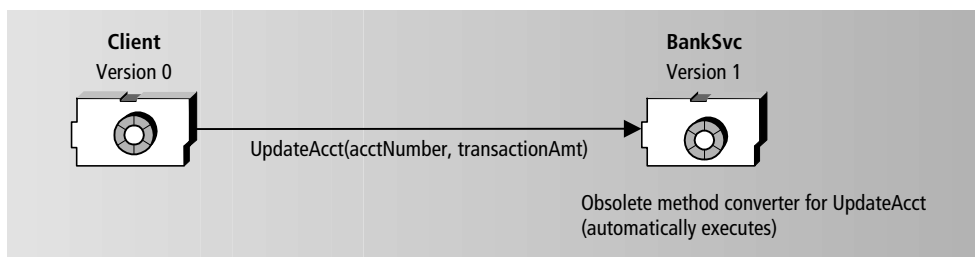


Figure 13-9 shows the opposite situation: what happens if an old client should invoke UpdateAcct on server that has been upgraded. The client has invoked a method (UpdateAcct with two parameters) that is now obsolete on the server, so the server automatically executes the obsolete method converter for that method. The client is unaffected by the fact that the server is changed.

**Figure 13-9** Caller is Older: Obsolete Converter Executes on Callee

In most upgrades, ultimately all client and server partitions are upgraded. When no older partitions are left, the converters are simply never invoked. They can be modified to delete references to the older class versions, or simply deleted altogether.

## Performing an Interoperable Upgrade

This section describes how to perform an interoperable upgrade. To determine whether this upgrade procedure will meet your needs, refer to [“About Interoperable Upgrades” on page 462](#) and to [“Changes Allowed Between Upgrades” on page 459](#). The table in the latter section lists changes that can be made in an interoperable upgrade.

In iPlanet UDS Release 2, this same upgrade approach is described as “Runtime Compatibility.”

---

**CAUTION** When performing an interoperable upgrade, it is your responsibility to ensure that the newer partitions are interoperable with the older partitions. If they are not interoperable, the system may function incorrectly and in unpredictable ways.

---

## Summary of Upgrade Steps

The upgrade steps for an interoperable upgrade simply entail making the desired changes to the application while ensuring interoperability. Then, as for any application, you make a new application distribution and install it on selected nodes. The application developer’s primary responsibility during this upgrade is to assure that all changes contained in the new distribution preserve interoperability.

► **To upgrade a deployed application using a new application distribution**

1. Make allowable changes to the application.

You may change a single class only, or you may change several classes in several projects. See the next section for more information on changes that you can make to the class components and application logic.

2. Make a new application distribution (see [“Making the Distribution” on page 477](#)).
3. Install new partitions on selected clients and servers.

---

**NOTE** The only visible difference between older and newer partitions is the date of the distribution. Every partition displays its build date when it starts. There is no other identifier that distinguishes older partitions from newer partitions, other than the dates on the distribution files.

---

## Changes Allowed in an Interoperable Upgrade

In partitions that are deployed, class definitions are fixed. Therefore, you cannot change a project in a newer partition in such a way that it will conflict with the same project on an older partition.

### Using Distributed Object References

Communication between partitions may fail or result in unpredictable behavior if any class component is unknown or changed so that it becomes incompatible with its definition in another partition. If you make many changes to an application, you should keep track of which classes are used by which partitions.

To ensure that there is no communication between the new partition and old partition involving anything unknown to the old partition, follow these basic rules:

- Do not post a new event on an object in the old partition.
- Do not invoke a new method on an object in the old partition.
- Do not get or set a new attribute on an object in the old partition.
- When you are passing a parameter in a method to an older partition, the class of the parameter (and all the classes of all its attributes, their attributes, and so on) must exist in the old partition. So, even though you can add new classes to the new partition, you cannot pass objects of new classes to the old partition.

## Adding New Attributes to Objects that are not Distributed

You can add new attributes to a class without the distributed property in the new partition when the class defines an object that is passed to or from the old partition. However, when iPlanet UDS passes the object to the old partition, it strips off the new attributes. This ensures that the object looks the way the old partition expects it to. Then, if the object is passed back to the new partition (as a return value, event parameter, or output parameter), iPlanet UDS sets the new attributes to their default values (the values set for them in the Init method for the object). Thus, on return from the old partition, the new attributes will always have their default values. You must take this into account when you write your code.

## Updating Window Classes

If a class is used *only in the new partition*, you can change that class freely. A good example is a window class. Because a window class is typically used only within the client partition, you can change the window class, even change method parameters, attribute types, and so on, without causing compatibility problems between partitions.

However, you must be very careful when you do this. As stated earlier, when you pass an object to an old partition, the old partition must have the correct definition not only for the class of that object, but for all its attributes, all their attributes, and so on. Communication between the partitions will fail if any class is unknown or changed. And it may be very difficult for you to keep track of which classes are used by which partitions.

## Making the Distribution

When you make a new application distribution, you make it for all application partitions; you cannot make a distribution for selected partitions. Then you install a subset of the new partitions on the nodes that you wish to upgrade, and upgrade the other nodes later.

To make a new distribution that is interoperable with a previous distribution, you can make the new distribution from either:

- the original repository (where you made the old distribution).  
If you make the distribution from the original repository, you can make a partial distribution, which requires slightly less time.
- a different repository than the original one.  
If you use a different repository, then you must make a full distribution.

If you make the new distribution from a different repository, you must first export the projects from the original repository with their IDs. Note that this is not the default way to export projects. To export the projects with their IDs, you must use the **ExportPlan** or **ExportWorkspace** commands in Fscript, with the **ids** option. Then you can safely transfer the projects from one repository to another, and make the distribution from the new repository.

If you move the projects to a new repository, you should not make parallel changes to both repositories. It is difficult to keep your changes synchronized, and you cannot merge the changes.

► **To make the distribution for the updated, interoperable application**

1. Use the repository where the original distribution was made, or make sure the projects were exported from that repository with their IDs.
2. In the Partition Workshop, use the Make Distribution command. See *A Guide to the iPlanet UDS Workshops* for information on this command.

You may select either a full or partial make, subject to the normal requirements for making distributions (for more information, see *A Guide to the iPlanet UDS Workshops*).

## Using Compatibility Levels to Upgrade

A compatibility level upgrade is required when you must make changes to an application that are not compatible with the current release of the application. A compatibility level upgrade always requires an increase in the application's compatibility level (as from CL0 to CL1). You are essentially creating a new application. To determine whether this is the upgrade procedure that meets your requirements, refer to [“Changes Allowed Between Upgrades” on page 459](#).

## Summary of Upgrade Steps

The steps for performing a compatibility level upgrade are the same steps for writing and installing any iPlanet UDS application. They are very briefly summarized here. For more detailed instructions, refer to *A Guide to the iPlanet UDS Workshops*.

► **To upgrade a deployed application using a new application distribution**

1. Modify the application. All changes are allowable.
2. Increment the compatibility level. Use the File > Properties command after opening the main project in the Project Workshop.

Refer to *A Guide to the iPlanet UDS Workshops* for information on this command

3. Make a new application distribution.
4. Install new partitions on desired clients and servers.

You must update *every* client that you wish to use the new level of the application; a client partition at an older (lower) compatibility level *cannot* access a server at a higher compatibility level.

## Using New Compatibility Levels of Libraries and Shared Service Objects

Compatibility levels are also associated with iPlanet UDS libraries and applications that contain service objects that are shared by multiple iPlanet UDS applications. The compatibility levels of an application, its supplier libraries, and its reference partitions need not be the same. However, when a distribution is made, iPlanet UDS notes the compatibility level of each shared resource used by an application so that it can subsequently enforce which applications may communicate.

Libraries and shared applications may be upgraded on different schedules than your application. As long as your application has access to the shared resources that were available when your application distribution was made, you need not automatically upgrade at the same time that the shared resources upgrade.

However, if you want your application to take advantage of new features and changes made to these shared resources, then you must make a new distribution of your application, so that iPlanet UDS will update the compatibility levels of the shared resources.

Note that the compatibility level of your application need not necessarily increment. Rather, the types of changes you make determine whether you must change the compatibility level of your application:

- You need not raise the compatibility level of your application if you make no changes or the changes you make are interoperable (shown in the third column of the table in *“Changes Allowed Between Upgrades”* on page 459).
- You need not raise the compatibility level of your application if you use class versions and converters.

However, if you do not meet these conditions, then you must raise the compatibility level for your application.

➤ **To use a new compatibility level of a library**

1. Import the library into your development repository. (See *A Guide to the iPlanet UDS Workshops* for more information.)

The new compatibility level of the library will overwrite the previous level.

2. Modify your application as necessary or desired to use the new library.
3. Make a new distribution for your application using the Partition Workshop.
4. Deploy the new release of the application.

➤ **To use a new compatibility level of a service object**

1. Obtain or make the new distribution for the application that contains the service object.
2. Repartition your application to use the newer service object using the Partition Workshop.
3. Make a new distribution for your application.
4. Deploy the new release of the application.



# Using Class Versions and Converters for a Rolling Upgrade

This section describes how to perform a class version upgrade. Class version upgrades are appropriate only for applications that have extremely critical availability requirements (known as “7 x 24”). Your application may meet this criteria in whole or in part: an application might have some services that must be available without fail as well as services that can temporarily interrupt service. The procedure in this section should be used only when upgrading servers of the former type.

This type of upgrade allows you to make substantive changes to a deployed application and also to install newer partitions while the application is running. It does not require you to change the compatibility level of your application, but you must use class versions and write converters to guarantee interoperability between older and newer partitions. For more information regarding whether this is the upgrade procedure you are interested in, refer to [“Changes Allowed Between Upgrades” on page 459](#).

To see a list of which changes are allowed during a class version upgrade, refer to [“About Class Version Upgrades” on page 465](#).

This upgrade procedure works with both standard and compiled partitions.

## Planning a Class Version Upgrade

Planning is a critical step for a class version upgrade. You should identify exactly which changes you will make to the application, and you should plan when you will upgrade which partitions, and in what order.

It is very helpful to identify which partitions reference which projects (and therefore, which classes). When used for the current application, the Fscript command **ShowApp** lists all projects referenced by that partition.

By using this command, you can determine whether you can update *all* partitions that reference a class that you need to modify (in which case you need not use converters), or whether you cannot update all partitions and so will need to write converters.

## Special Requirements for High Availability Servers

If your application has high availability servers, you must meet the following requirements, which are expected and typical for these types of applications:

**Two locations for old and new application installations** A class version upgrade requires at least two server nodes, each with its own FORTE\_ROOT directory structure. This allows you to keep both old and new copies of the application, because image repositories and executables for a deployed application are stored at the same default location relative to each FORTE\_ROOT.

**At least two replicates of high-availability servers** Each high availability server partition that is to be upgraded must be installed on at least two nodes in the current deployment. This allows at least one server to be available to serve clients, while shutting down one or more other server processes for upgrade.

**Adequate failure recovery** The application must be configured to tolerate runtime failures, and handle failure recovery. This allows the shut down of any server partition without application failure during the upgrade procedure. This can be achieved, for example, by doing the following:

- specifying server dialog duration as “Message” for transparent failover
- coding clients to handle runtime exceptions in case of failover

## Summary of Upgrade Steps

The following steps describe the additional application development steps required to upgrade an application using converters and class versions. Each step is described later in this chapter.

### ► **To upgrade a deployed application, using class versions and converters**

Perform Steps 1 to 5 in the iPlanet UDS Workshops. Steps 1 to 3 can be performed in any order.

1. Make allowable changes to class components and write corresponding converters (see [“Updating Classes and Writing Converters” on page 483](#)).
2. Increment the version number of each class that you modify (see [“Using Class Version Numbers” on page 492](#)).
3. Test converters (see [“Testing Converters” on page 493](#)).
4. Make an application distribution (see [“Making a Distribution” on page 493](#)).

Perform the following step in the deployment environment:

5. Update client and server nodes with the new partitions (see [“Installing and Starting Updated Partitions”](#) on page 493).

Steps 4 and 5 are largely the same as for any iPlanet UDS application. For further information on these steps, see *A Guide to the iPlanet UDS Workshops* and *iPlanet UDS System Management Guide*.

## Updating Classes and Writing Converters

A class version upgrade allows you to modify class definitions significantly. See [“About Class Version Upgrades”](#) on page 465 for list of which converters are required by which changes.

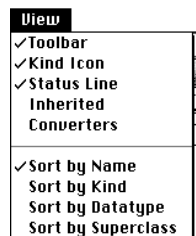
As you make changes to the classes, you should keep track of which changes require converters, so that you create and define all necessary converters. You must also update all references to the changed component. That is, if you change a method signature, you should edit all other methods in which that method is invoked, so that each invocation uses the new signature. Finally, you should raise the class version number for each class in which you have made changes.

### Viewing Converters

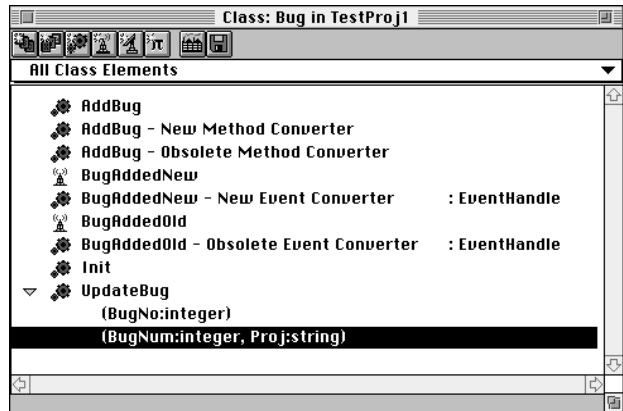
By default, converters do not display in the workshops. You can request that converters be displayed for a given class in the Class Workshop.

#### ► To see all converters defined for a class

1. In the Class Workshop for a class, select the command View > Converters. Using this command toggles the display of all converters for a given class.



The drop list in the Class Workshop also affects whether converters are shown. If either “All Class Elements” or “Methods” is selected, converters are displayed. Note that event converters do not appear when “Events” is selected, because event converters are actually methods. Converters appear as in the Class Workshop below.



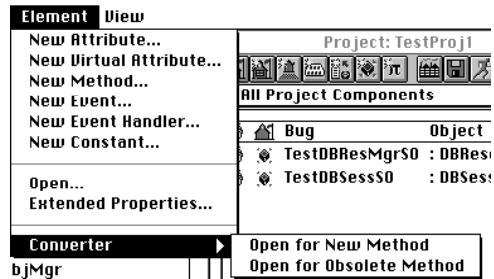
► **To see the text of an individual converter**

1. Enter the Class Workshop.
2. Toggle on View > Converters.
3. Double click on the converter.

or

1. Enter the Class Workshop for the class that has converter methods. Select the method or event for which you wish to see converters.
2. Select Element > Converter.

3. If you have converters defined for the current method or event, you will see the options:



Choose either option depending on which converter you want to see. If you have no converters defined, the slide off menu shows “Create” instead of “Open.”

## Guidelines for Writing Converters

You write the converters at the same time that you update the class by modifying its components. Which converters you must write, and their content, depend on the changes you make to the class. Note that you should create some converters *before* altering a method or event (see below for more information). For a review of which changes require which converters, see the table in [“About Class Version Upgrades” on page 465](#).

Since all converters are methods, you use the Method Workshop to write a converter. The following information is useful when writing any converter:

- You always define converters in the newer version of the class.
- The name and signature of every converter are automatically generated. They can only be changed by changing the corresponding method or event. The converter code must be provided by the developer.
- *You must define an obsolete converter before you either change a method signature or delete a method.*

If you do not define an obsolete converter before you either change a method signature or delete a method, then a valid converter signature cannot be generated. (Changing a method’s signature is the same as adding a new method (with the newer signature) and deleting the method with the former signature.)

- Before you delete a class you must create an obsolete method converter for every method on the class. The deletion of a class is treated as a class in which all methods have been deleted.
- You can use **case** statements in converters. If the converter converts between multiple versions, **case** statements help to isolate the logic that applies to each class version. The following code fragment from a new converter anticipates the invocation of the method from an older version caller (version 0 is the oldest possible version):

```
method new NewMethod(param1 : integer)
begin
  case classversion is
    0 do
      // take action equivalent to this method when calling
      // an older version of the object
    end case;
end
```

- The new **classversion** TOOL key word shows the class version of the partner object (that is, the object that invoked the new or obsolete method associated with the converter). Both new and obsolete converters reference the older class version.

For example, with two versions of a class (0 and 1), a new converter, which executes on the *newer caller*, would refer to the called object, which is class version 0. An obsolete converter, which executes on the *newer callee*, would refer to the calling object, which is classversion 0. Referring to [Figure 13-8 on page 474](#) and [Figure 13-9 on page 475](#) may help you see this.

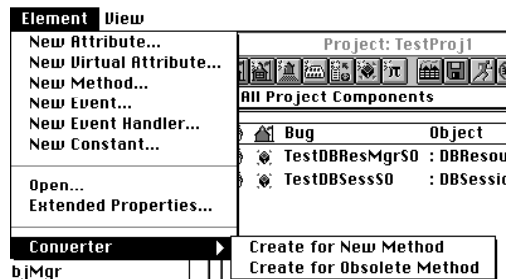
- A converter can invoke any method associated with the current (newest) version of the class.
- A converter cannot directly invoke other converters, but may cause other converters to be invoked as a result of referencing a component for which a converter exists.
- If a subclass inherits a class component, it also inherits any converters on that component.
- You write converters for overloaded methods just as for any method.
- You cannot override a converter that has been inherited by a subclass. However, if a subclass overrides an inherited method, you can write a converter for the overriding method.

## Writing Method Converters

When you write a method converter, you write code that will handle a different version of the class invoking the method (that is, the version of the remote method is different from the version of the local method). You begin to write converters using the Class Workshop.

### ► To create a method converter

1. In the Class Workshop, highlight the method requiring a converter. Select **Element > Converter**. If the current method has no converters defined, you will see the options:



2. After you choose the appropriate type of converter, the Method Workshop opens. The name and parameters of the converter are automatically derived; you cannot change them.
3. Write the converter code using TOOL.

When you write a new method converter you can code it in terms of a similar previous method, code it to raise an exception, or do nothing. A new converter always runs on the newer *caller*, and therefore each reference to “self” in the new converter executes as a separate call to a distributed object reference on the older callee.

A new converter can only reference attributes and methods that exist on the older object, and other new methods for which there is a new method converter. Any attribute referenced in the converter is a remote attribute access (which therefore executes on a copy of any non-anchored object attribute, following the normal rules of remote attribute access). References to attributes not in the old version will cause unpredictable results at runtime.

If the distributed call requiring a new method converter execution is invoked on a remote message dialog duration service, there are additional implementation considerations, particularly if the service is load balanced. While these considerations are strictly due to normal iPlanet UDS distributed processing, in the context of new converter execution the remote calls are implicitly invoked, and may require additional planning.

For more information about Message dialog duration, refer to [“Assigning a Dialog Duration to Service Objects” on page 402](#).

Each reference to “self” in the new converter method (either explicit or implicit), which accesses a method or an attribute of the service, is invoked using a remote reference to that service. For Message dialog duration calls (which operate only for the extent of a single method call or attribute access), each subsequent reference may in fact resolve to a *different* instance of the service, transparent to the client.

If the service is load balanced, then each reference may resolve (that is, be routed) to a different replicate of the load balanced set. Or (if not replicated for load balancing) potentially, a failover could occur between distinct references to the service, which the client would not detect.

The implications of these example situations are that what is intended to be a single unit of work occurring on a single remote server instance may possibly in fact result in calls to more than one server instance.

Also, you should anticipate behavior when invoking an explicit transaction block within a new converter method that invokes remote references. In particular, note that transactions *do not propagate* to a remote Message dialog duration service (see [“Dialog Duration and Error Handling” on page 404](#)).

Finally, with respect to state information in the remote partition, you should take care when directly accessing attributes on the remote server from the new converter. In particular, remember that attribute references which are not distributed or anchored will operate on a *copy* of the attribute.

These issues are generally not a concern for other dialog durations, (whether load balanced or not), because a unit of work contained within a transaction is always processed against exactly one server instance, and the client should detect any failure for the duration of that processing.

An obsolete method converter always runs on the newer *callee*, and therefore executes on the actual object (instead of a distributed object reference to the object). An obsolete method converter can reference any attribute or method that exists in the current (newer) version of the class. References to attributes not in the newer version will cause unpredictable results at runtime.

To see examples of method converters, see [“The Banking1-2 Example” on page 472](#).



## Writing Event Converters

You always write event converters in pairs. You write a new event converter on the new event and an obsolete event converter on the old event. Remember that event converters are actually methods.

The following guidelines apply when writing event converters.

- All attributes that are referenced in an event converter must exist in the *both* class versions.

---

**CAUTION** In an event converter, references to attributes not in both class versions can cause unpredictable results at runtime.

---

- Every event converter should be able to deal with the case of zero/NIL parameter values. Either event converter may be invoked when the values of all of the parameters are zero. This is normal and occurs when the event is being registered for. In this case, the converter should simply return the replacement event with zeros or NIL as the parameter values; the values are simply ignored.
- Raising an exception in an event converter is not recommended; event converters can be invoked outside of normal application flow in which case an exception has no effect.
- Event converters should generally invoke read-only operations, although iPlanet UDS does not enforce this.

Given that the order of partition upgrades is not deterministic (that is, a group of client/server partitions might be upgraded in any order), it can be difficult for an iPlanet UDS application developer to anticipate when either a new or an obsolete event converter will be executed. Event converter execution can occur upon invocation of an event posting/registration, to or from a new or old partition; as such, you would normally not want updates to occur indeterminately against any set of critical data.

- At a minimum, you should always code event converters to explicitly return the exact event which is complimentary in its event converter pair.
- All application code which runs in the same partition with the *new* class version should *not* reference the *old* event; it should only post and register for the *new* event directly. (Only a converter should ever actually reference an obsoleted event).

- When you create a new/obsolete event converter pair, you should change all references in your (new) code to access the new event (that is, all classes, projects, and so on which comprise the application). Otherwise, runtime problems might occur if a remote (distributed) invocation causes a converter to execute, but the (new) code still references the “obsoleted” event: a different event may get posted than is intended to be posted/registered for.

➤ **To replace an event and write the associated event converters**

1. Create an obsolete event converter for the event to be replaced. Code the converter method to return the newer event (with the newer parameters).

The parameters for an obsolete event converter are automatically the same as the parameters for the (older) event. The obsolete event converter method should return an object of class `eventhandle`, specifically, the newer event.

2. Create the new event to replace the old event. The new event will have a different name and a different set of parameters.
3. Create a new event converter for the new event. Code this converter method to return an object of class `eventhandle`, specifically, the old event.

Assume that a bug tracking application defines an event called `BugAddedOld`, that has one parameter, `BugNo`. Now that the company has grown, bugs must be associated with different projects, so we want to define a new event `BugAddedNew` to add one parameter called `Proj`.

➤ **To replace the event `BugAddedOld` with a new event `BugAddedNew`**

1. Create an obsolete converter for the old event: `BugAddedOld`. It should return the new event, using the new event’s parameters, and substituting reasonable default values (as in 00 for an undefined project).

```
method obsolete conv.BugAddedOld(bugNo: integer) : eventhandle  
return BugAddedNew(bugNo, NIL);
```

2. Define the new event as `BugAddedNew (bugNo : integer, proj : string)`

3. Create a new event converter for the new event. It should return the old event, using the old event's parameters, stripping away parameters that only apply to the newer event.

```
method new conv.BugAddedNew(bugNo: integer, proj : string) : eventhandle
return BugAddedOld(bugNo);
```

## Modifying Converters

Whenever you increment a class version you will either create new converters or modify existing converters. The name and signature for a converter are automatically generated (and updated if you change the underlying method). You write and modify a converter's code just as you do with any other method.

To modify a converter you use the Class Workshop. After you double-click on the converter name the Method Workshop opens with the converter code displayed for you to edit.

If you increment the version for a class, you need not necessarily edit every converter to add a new block in the case statement for that version; you need only edit the converters for which there is a difference in the classes, as shown in the table in [“Changes Allowed Between Upgrades” on page 459](#).

## Deleting Converters

You can delete converters when they are no longer necessary. You should not delete a converter until you are sure that all versions referenced by the converter are obsolete and not referenced by any partition in an application.

Since converters are simply methods, you use the Class Workshop to select the converter you want to delete, and use the Element > Delete command to delete it. Because converters are separate methods that always include “converter” in their name, it is easy to identify and delete them.

Deleting a method also deletes its new method converter automatically. Any associated obsolete method converter is not deleted.

Obsolete converters must be explicitly deleted; they are not deleted when you delete the associated class component.

## Using Class Version Numbers

A version number is a runtime property of a single class. A version is zero (0) by default; note that the class versions for applications running on iPlanet UDS Release 2 is always 0. You can set the version using the Class Properties dialog. A class can have a different version level from its parent class and does not inherit the version from its parent class.

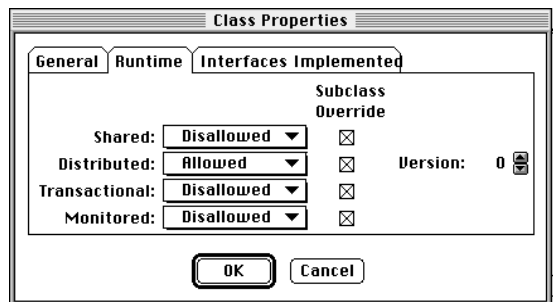
Some general guidelines for using class version numbers follow:

- You can set the class version number for any class.
- While you can have any number of versions of a class, it is easier to manage if you keep the number of versions to a minimum, since converters are potentially required between versions.
- Class version numbers are independent of one another. If you increment one class version from 1 to 2, you need not increment any other class. However, it may be helpful to use the same new version number for changes that are related or made at the same time.
- Once you have deployed a class with a particular version, you should not decrement the version for that class. Subsequent changes to that class should be marked with a higher version number.

You can change the version number at any time while updating a class.

### ► To set a class version number

1. Open the Class Workshop for the class that you intend to modify.
2. Select the File > Properties command. You will see the Class Properties dialog, as shown below. Go to the Runtime tab page.



3. Increment the version number for the class.

## Testing Converters

You should keep a copy of the distribution that is currently deployed to use to test converters. One way to test obsolete converters is to set up a small, parallel system in which you can run multiple distributions of an application and work through the stages of an upgrade. In fact, a test system allows you to more thoroughly test all converters by repeatedly staging various test upgrade scenarios.

## Making a Distribution

You will make one or more application distributions in order to test your changes and converters, and you will make a final distribution to use for the actual upgrade. You should make each distribution from the same development repository, the same workspace, and into the same target environment as the previous application. (It is possible to use a different repository if you follow the guidelines in [“Making the Distribution” on page 477.](#))

## Installing and Starting Updated Partitions

If you can shut down a partition—as you usually can with client partitions—then you can simply replace the older partition with the newer partition. However, if you cannot shut down a partition, as may be true for server partitions, especially high availability servers, then you must install the new distribution on a different (logical) node than the existing, currently running server partition.

As with any application upgrade, you may want to save the distribution and/or installed distribution (userapp directories) before upgrading, so that the upgrade can be backed out if necessary.

---

**CAUTION** Never replace an older installed application file executable or image repository with a newer file while the application is running on any individual node. This is an operating system restriction, and results are unpredictable if you attempt to do so.

When you install an application on a node, you install all partitions for that application and that node at the same time; similarly, to *upgrade* an application on a node, you upgrade all application partitions on the node. Thus, you should shut down *all* partitions on the node for an application before installing the new partitions.

---

➤ **To upgrade a client partition**

1. If necessary, shut down the client partition(s) of the application on the node.
2. Install the new client partition.
3. Restart the client partition(s).

You can follow the previous procedure for any server that does not have high availability requirements.

➤ **To upgrade a high availability server partition**

---

**CAUTION** Do not shut down a high availability server partition *if you cannot guarantee that at least one server partition is running that will be used for failover*. You must make sure that an adequate number of replicate server partitions are running to meet your failover and load balancing requirements.

---

1. If the partition is currently running, shut down the partition and process. Use the partition agent in Escript or Econsole; locate the agent using the **FindSubAgent** command, invoked from the application agent level.

(A shutdown may not be necessary—even if the partition has high availability requirements—if multiple replicates are currently running to service clients.)

2. In Escript, invoke the **InstallApp** command on the Node agent.

This command installs all partitions on this node for the new application version.

3. In Escript, and at the partition agent for each partition on the node, invoke the **start-up** command. You can use any start-up method when you perform a class version upgrade. If you use autostart, however, only one replicate is started.

To see what steps will occur at each install, you can view the `InstallationSteps` instrument.

There is no specific order in which you should upgrade partitions, whether client or server. This is because all applications and their business contexts are unique, especially complex applications. It is also true because the relationships between partitions are complex, and many server partitions in fact act as both client and server, and may have circular references among servers. However, there are two guidelines that you should understand:

- If you can assure or require that certain partitions are upgraded before others, you may be able to simplify the upgrade process somewhat. If you have a good understanding of your application, you might be able to dictate an optimal order for upgrading partitions, and thereby reduce the number of converters that are required. For example, if you know that all servers will be upgraded first and that some can be upgraded simultaneously, you may be able to omit writing some new method converters. In this case, the order of upgrade should not be deviated from.
- If you want to guarantee interoperability between partitions no matter what order partitions are upgraded, then you should write all converters that might possibly be invoked. This approach may require slightly more coding, but also allows more fault tolerance and flexibility in the order of upgrade.

## Using Failover

It is assumed that you will use the failover feature when performing a class version rolling upgrade, because this type of upgrade is intended for applications having high availability requirements.

There are no special requirements when using failover. The only difference from normal failover is that the server replicates are not identical replicates. Clients will failover from an older server partition to a newer server partition.

## Using Load Balancing

You can use load balancing while performing a class version upgrade. If you use load balancing for a server partition that exists both in old and new form, then you must use failover to provide at least two router partitions: one router to load balance requests for the older server partitions and one to balance requests for the newer server partitions.

A router partition, once started for a server partition of a given distribution, only balances requests for server partitions of the same distribution as the original partition.

The following procedure assumes that the application is running, and the router for the older server and its replicates is already running.

### ► **To use load balancing for a server having multiple versions**

1. Start the newer version of the server, and as many replicates as you desire.
2. Manually start a router partition; it will automatically load balance requests for the newer server.

## Recording Information About the Update

It is highly recommended that you record the changes you make during a class version upgrade. While converters are designed to ease the rolling upgrade process, they do add some additional maintenance burden.

If you make no notes on your own, iPlanet UDS tells you the compatibility level of a distribution and date it was made. It is not possible to determine which version of a class is contained in a given partition from a distribution. So you should record the following information whenever you update an application:

- for each partition, which classes changed, and the version number of each class
- which nodes installed the new partitions

To see what projects are used by an individual partition, you can use the Fscript command **ShowApp**. Note that this same information is not available through the Partition Workshop.

## Removing Versions of Classes

When you know that a version of a class is no longer needed, because there is no remaining client or server partition of that version, then you can remove all references to that class version. This is easy to do; simply edit the converters to remove the sections which refer to those class versions.



# TOOL Reflection Classes

This chapter introduces the TOOL classes that implement reflection. A running program can use reflection to examine or to change objects that are running in a local or remote partition.

A detailed description of each reflection class is provided in online help.

## The Power of Reflection

The iPlanet UDS runtime provides a set of reflection classes that you can use to get information about user-defined and system classes and interfaces at runtime. You can also get this information for instances of a class, whether the object is local or distributed. In addition to getting information about a class or interface, you can use reflection classes to retrieve current attribute values, to set new attribute values, or to invoke methods on an object.

Because reflection allows you to examine classes, interfaces, and their members, you can use it to develop a variety of tools and other software:

- Development tools like debuggers, specialized class or object hierarchy browsers, testing tools, profiling tools, structural analysis tools, trace facilities, and so on.
- Framework developers can use reflection to build components from user-defined classes without prior knowledge of their structure.
- Code that implements persistence can use reflection to determine at runtime the structure of objects to be stored.

- External systems that request a service whose particulars are not known until runtime can use reflection to determine which implementation of an interface can handle the requested operation.
- System management tools can use reflection to get information about objects at runtime and to test the behavior of these objects (by changing attribute values or invoking methods) without having access to the source code.

This chapter introduces the TOOL reflection classes that you use to develop tools like the ones described above. Each reflection class is described in detail in the online help.

This chapter includes small code samples that illustrate the use of the TOOL reflection API. For more extensive examples, see [“Reflection Examples” on page 510](#).

## Learning About Reflection

The TOOL reflection API is closely modeled on the Java reflection API. Therefore, literature describing the use of the Java APIs can help you learn how to use the TOOL reflection classes. For Javasoft’s documentation about reflection, see <http://java.sun.com/products/jdk/1.3/docs/guide/reflection/index.html>

## Restrictions

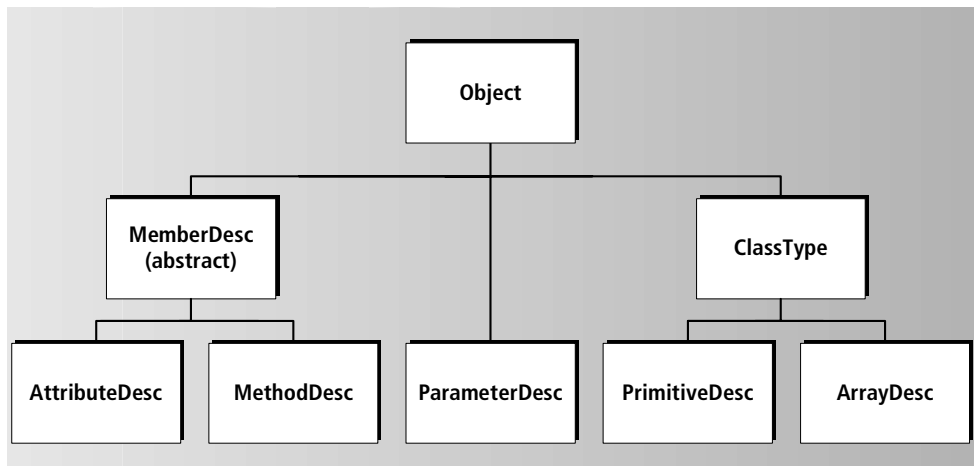
The use of reflection classes is limited in the following ways:

- You cannot use reflection classes to access information about parameter names or about the events, virtual attributes, exceptions, properties, or constants defined for a class or method.
- If the current attribute is a virtual attribute, you cannot use reflection to get or set the values of this attribute for an object.
- Some data types cannot be accessed using the Get and Set methods on the AttributeDesc class. These data types include C-style array, enum, pointer, struct, typedef, and union. For more information, see [“Getting or Setting the Value of a Primitive Type” on page 506](#).

# TOOL Class Reflection

The TOOL classes that support reflection and the inheritance hierarchy that determines their relation are shown in [Figure 14-1](#).

**Figure 14-1** TOOL Reflection Class Hierarchy



Using TOOL reflection classes, you can get information about

- the type of a class
- the methods defined for a class or an interface
- the attributes of a class

At runtime, you can also use reflection to get and set the value of an attribute for a particular object, or to invoke a method on that object. You can retrieve information about an object whether it is local or distributed.

The following table lists and describes the TOOL classes that enable class reflection:

Class	Description
ClassType	Provides methods that you can use to obtain references to AttributeDesc or MethodDesc objects.
ArrayDesc	Provides methods that you can use to get information about array data types.

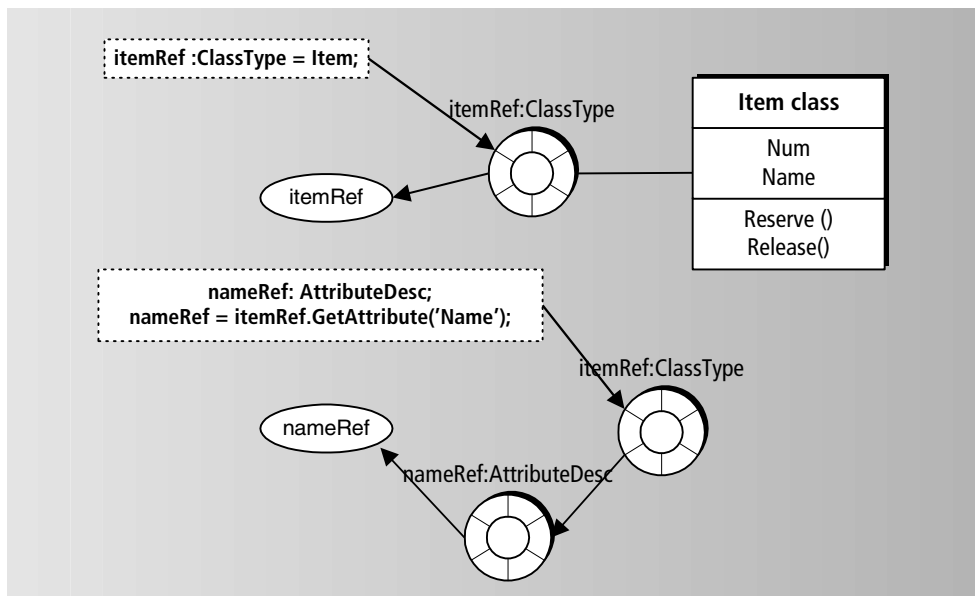
Class	Description
PrimitiveDesc	Provides methods that you can use to get information about primitive (simple) data type.
MemberDesc	An abstract superclass of the MethodDesc and AttributeDesc classes.
MethodDesc	Provides methods that you can use to determine the signature of a given method or to invoke a method on a particular object.
AttributeDesc	Provides methods that let you get or set an attribute value on a particular object.
ParameterDesc	Provides methods that you can use to determine the data type and parameter passing mechanism of a given parameter.

You do not have to link against any particular library to use reflection classes: they are an integral part of the TOOL Framework library.

## Accessing Reflection Objects

In general, you do not create instances of reflection classes yourself. A `ClassType` object is automatically created for every user-defined (and system) class at compile time. Instead, you create a reference to a `ClassType` object for the class or interface that interests you, and then use the methods defined for `ClassType` to obtain objects that reflect the attributes and methods of that class. For example, you can use the `GetAttribute` method of `ClassType` to get an instance of the `AttributeDesc` class. You can then use the **Get** methods defined for the `AttributeDesc` class to get additional information about the attribute.

**Figure 14-2** shows how you obtain a reference to a `ClassType` object associated with the class `Item`, and then a reference to an `AttributeDesc` object that reflects information about the `Name` attribute.

**Figure 14-2** Using ClassType Objects to Enable Reflection

The only time you need to create an instance of a reflection class is when you need an `ArrayDesc` object in order to specify parameters to `ClassType.GetMethod()`, and `ClassType.GetDeclaredMethod()`. This case is discussed in [“Working With Methods”](#) on page 508.

## Getting Information About a Class or Interface

Each class or interface that has been partitioned is reflected by a single `ClassType` object. The `ClassType` class defines several methods that return information about the reflected class, such as the methods or attributes defined for that class.

To obtain information about a class that interests you, you must first define a reference to the `ClassType` object that is associated with your class, using the following syntax:

```
variable_name : ClassType = class_name;
```

You can get information about user-defined classes as well as system classes. The following example shows how to create a reference to the `ClassType` object that reflects the `Employee` class:

```
empCTRef : ClassType = Employee;
```

If you don't know the class name but have an object that interests you, you can use the `GetClassType()` method of the `Object` class; for example:

**Code Example 14-1** Using the `GetClassType` method

```
anyClass: ClassType;  
anyClass = anObject.GetClassType();
```

Having obtained a reference to the `ClassType` object for the class, you access additional information about that class by calling methods on the `ClassType` object. For example, to get information about the `EmpName` attribute of the `Employee` class, you would write:

```
attName : AttributeDesc = empCRef.GetAttribute(name='EmpName');
```

You would then use methods defined for the `AttributeDesc` class to learn more about that attribute.

`ClassType` methods allow you to get

- a reference to the `AttributeDesc` object that reflects an attribute, or an array of references to `AttributeDesc` objects that reflect all the attributes of a class.

You can then use methods of the `AttributeDesc` object to get the value or type of a reflected attribute.

- a reference to the `MethodDesc` object that reflects the signature of a specific method, or an array of references to `MethodDesc` objects that reflect the signatures of all methods for that class.

You can then use the methods of the `MethodDesc` class to determine the method's return type, its parameter types, and its visibility. You can also use `MethodDesc.Invoke()` to invoke the method.

- an array of `ClassType` objects that reflect the interfaces implemented by a class.
- the name of a class, interface, or simple type
- the `ClassType` object that reflects the superclass of this class

You can also use the `ClassType.NewInstance` method to create a new instance of the current class.

Of course, in an application that uses reflection, you would probably access a class whose name (or whose member names) is not hard coded in your application. Rather, you may be passed a parameter that is an object whose class you are interested in, as in the following example:

**Code Example 14-2** Accessing class information about an object

```
method Browser.InspectAttributes (obj: Object)
begin
  cls: ClassType (obj.GetClassType());
  ats: array of AttributeDesc;
  ats = cls.GetAttributes();
  for att in ats do
    ... //get or set attributes
  end;
```

## Accessing Arrays

If the data type of an attribute, parameter, or return value is an array, such as Array of TextData, the type is reflected by an ArrayDesc object. The ArrayDesc class, a subclass of ClassType, provides methods and attributes that you use to retrieve information about array elements and their data types.

You might need to create ArrayDesc objects to specify parameters for the ClassType.GetMethod() and ClassType.GetDeclaredMethod(). In the following code fragment, you instantiate an ArrayDesc object to describe a formal parameter, formals[3], that is an array of IntegerData:

**Code Example 14-3** Using ArrayDesc to get information about array elements

```
formals: array of ClassType = new;
formals[1] = ClassType.StringPrimitiveDesc;
formals[2] = ClassType.IntegerPrimitiveDesc;
formals[3] = ArrayDesc(ElementType=IntegerData);
```

Use the ClassType.IsArray() method to determine whether the current class type is an array type.

## Accessing Simple Data Types

If the data type of an attribute, parameter, or return value is a TOOL simple data type, such as int or boolean, the type is reflected by a PrimitiveDesc object. The PrimitiveDesc class, a subclass of ClassType, has methods and attributes that let you specify or identify a particular simple data type.

You access the PrimitiveDesc object that reflects a simple data type by using the corresponding attribute of the ClassType class. These attributes are like static members in Java: you must access them using the following syntax:

**ClassType**.*attribute\_name*

For example, to reflect an integer, use the ClassType.IntPrimitiveDesc object; to reflect a char, use the ClassType.CharPrimitiveDesc object. The following code sample uses the PrimitiveDesc object that reflects a boolean to check whether the data type of a return value is a boolean:

**Code Example 14-4** Using PrimitiveDesc to determine the data type of a return value

```
tDataRef: ClassType = TextData;
isLowerMethod: MethodDesc = tDataRef.GetMethod
                                (name = 'IsLower', parameters = NIL);
if (isLowerMethod.GetReturnType() =
    ClassType.BooleanPrimitiveDesc)
then
    task.Lgr.PutLine('IsLower:boolean');
end;
```

## Working With Attributes

The AttributeDesc class provides methods that you use to get and set the value of the current attribute for a particular object or a particular primitive data type. Note that reflection allows you to get or set the value of any private attribute, which would normally be restricted to the class defining that attribute.

### Getting the AttributeDesc Object

To get the AttributeDesc object for an attribute, you can use any of the following methods of the ClassType class:

- GetAttribute
- GetDeclaredAttribute



- `GetAttributes`
- `GetDeclaredAttributes`

For additional information about these methods, use the online help for the `ClassType` class.

## Determining the Data Type of an Attribute

Once you have obtained a reference to an `AttributeDesc` object for an attribute, you can use the `GetType` method to determine its data type. This method returns a reference to a `ClassType` object that reflects the class, interface, or primitive data type for the attribute. For example, assuming that we need information about a class declared as follows:

### Code Example 14-5 Getting the data type of attributes

```
Class Item inherits Object
    Num: integer;
    Name: TextData;
end;
```

We can use code like the following to print out the data type of two attributes, `Num` and `Name`:

```
cls: ClassType = Item;
att: AttributeDesc = cls.GetAttribute('Num');
task.Lgr.Putline(att.GetType().GetName());           //integer
att = cls.GetAttribute('Name');
task.Lgr.Putline(att.GetType().GetName());           //TextData
```

## Getting or Setting the Attribute Value on an Object

To get or set the value of the current attribute when the data type is a class or interface, use the `GetValue` and `SetValue` methods (of the `AttributeDesc` class), specifying an object whose class includes that attribute.

The following example demonstrates how you can retrieve an attribute value using the `GetValue` method on an `AttributeDesc` object named `att`:

**Code Example 14-6** Using `GetValue` to retrieve an attribute value

```
cls: ClassType = Item;
myItem:Item = new (Name = 'New Item');
att: AttributeDesc = cls.GetAttribute('Name');
name:TextData = TextData(att.GetValue(myItem));
task.Lgr.Putline (Name); // "New Item"
```

### *Using DataValue Objects*

To get or set an attribute value using a `DataValue` object, use the `GetValue` method or the `SetValue` method. If the type of the value is a primitive type, the `GetValue` method automatically instantiates a `DataValue` object that contains the value; for example, an `IntegerData` object to contain an integer. Similarly, the `SetValue` method extracts the primitive data value from the specified `DataValue` object to set the value of an attribute with a primitive data type. Note, however, that there are methods defined for the `AttributeDesc` class that are specific to primitive types. The section [“Getting or Setting the Value of a Primitive Type”](#) introduces these methods.

For detailed information about the `GetValue` and `SetValue` methods, see the online help description of the `AttributeDesc` class.

### Getting or Setting the Value of a Primitive Type

To get or set an attribute value using a primitive type, use one of the `Get` or `Set` methods of the `AttributeDesc` class that is specific to the primitive type, such as `GetBoolean` to retrieve the value as a boolean. It is more efficient to use the specific `Get` or `Set` method than to create (or cause the system to create) a `DataValue` object with which to wrap your primitive value. The following table lists the `Get` and `Set` methods for each primitive type

<b>TOOL Data Type</b>	<b>Get Method</b>	<b>Set Method</b>
boolean	<code>GetBoolean</code>	<code>SetBoolean</code>
char	<code>GetChar</code>	<code>SetChar</code>
double	<code>GetDouble</code>	<code>SetDouble</code>
float	<code>GetFloat</code>	<code>SetFloat</code>

TOOL Data Type	Get Method	Set Method
i1	GetI1	SetI1
i2	GetI2	SetI2
i4	GetI4	SetI4
int	GetInt	SetInt
integer	GetInteger	SetInteger
long	GetLong	SetLong
short	GetShort	SetShort
string	GetString	SetString
ui1	GetUI1	SetUI1
ui2	GetUI2	SetUI2
ui4	GetUI4	SetUI4
uint	GetUInt	SetUInt
ulong	GetULong	SetULong
ushort	GetUShort	SetUShort

If you are not sure which primitive data type describes the current attribute, you can compare the `PrimitiveDesc` object returned by the `AttributeDesc.GetType` method with the `ClassType` attributes for the various `PrimitiveDesc` objects. For example:

**Code Example 14-7** Determining the primitive data type of an attribute

```
att: AttributeDesc = cls.GetAttribute('Value');
if (att.GetType() = ClassType.BooleanPrimitiveDesc) then
    b:boolean = att.GetBoolean(source = myObj);
else if (att.GetType() = ClassType.DoublePrimitiveDesc) then
    d:double = att.GetDouble(source = myObj);
...
else
    o:Object = att.GetValue(source = myObj);
end;
```

If the current attribute is a virtual attribute, you cannot get or set the values of this attribute for an object. To determine whether an attribute is a virtual attribute, use the `IsVirtual` method, described in the online help for the `AttributeDesc` class.

Certain `PrimitiveDesc` objects represent data types that cannot be accessed using the `Get` and `Set` methods on the `AttributeDesc` class. These data types include C-style array, enum, pointer, struct, typedef, and union. To test whether a `PrimitiveDesc` object represents a supported data type, use the `PrimitiveDesc.IsSupported` method, as described in online help for the `PrimitiveDesc` class. If you try to get or set an attribute value whose data type is not supported, the `Get` or `Set` method raises a `UsageException`.

## Working With Methods

You can use reflection to get a method's signature and to invoke a method. Note that reflection allows you to get information about and to invoke private methods, which would normally be restricted to the class for which the method is defined.

To get information about a method or to invoke a method, you must first get a reference to the `MethodDesc` object for that method. You can use the `ClassType.GetMethods` method to retrieve an array of references to `MethodDesc` objects for a particular class; for example:

```
allMethods : Array of MethodDesc = currCls.GetMethods();
```

You can then call the appropriate method of `MethodDesc` to obtain the signature of a particular method. Having obtained the signature, you can use the `Invoke` method of a `MethodDesc` object to invoke a method on an instance of the class.

You can also use the `ClassType.GetMethod` method to retrieve a reference to a `MethodDesc` object for a particular method if you know its parameter list. See online help for the `ClassType` class for more information about `GetMethods` and `GetMethod`.

Assuming the following class declaration:

### Code Example 14-8 Getting a method's signature and invoking the method

```
class Item inherits Object
    IsRegistered(name:string,
                num: integer,
                ids: array of IntegerData): boolean;
end;
```

You can get information about the `IsRegistered` method as follows:

```
...
cls:ClassType = Item;
formals: array of ClassType = new;
formals[1] = ClassType.StringPrimitiveDesc; //param = name
formals[2] = ClassType.IntegerPrimitiveDesc; //param = num
formals[3] = ArrayDesc(ElementType=IntegerData); //param = ids
meth:MethodDesc = cls.GetMethod('IsRegistered', formals);
```

## Getting Parameter and Return Value Information

When you have retrieved the `MethodDesc` object for the method of interest, you can get information about the parameters of the method using the `GetParameters` method and about the return value using the `GetReturnValue` method of the `MethodDesc` class.

The `GetParameters` method returns an array of references to `ParameterDesc` objects that reflect the parameters of the current method. You can then use the `GetType` and `GetMechanism` methods on the `ParameterDesc` class to determine the type of each parameter and how it is passed to the method.

## Invoking the Method

Finally, you can invoke the method represented by a `MethodDesc` object using the `Invoke` method and passing a particular object whose method you want to invoke. Remember that the type of the object must be the same as or a subclass of the class that declared the current method, represented by the `MethodDesc`.

The following example shows how you can invoke a method after you have retrieved the `MethodDesc` object that reflects the method.

```
-- cls is a ClassType object.
formals: array of ClassType;
actuals: array of Object;
formals = ...;
actuals = ...;
currMethod : MethodDesc = cls.GetMethod(name = 'display',
    parameters = formals);
rvalue : Object = currMethod.Invoke(source = windowToOpen,
    parameters = actuals);
```

The value returned by the `Invoke` method is the return value for the invoked method. The `Invoke` method also promotes any exceptions that are raised by the invoked method.

## Reflection Examples

This section provides two sample code listings that demonstrate how you use reflection to implement a simple object inspector and a simple class browser.

### Object Inspector

The `ObjectInspector` class, shown in the listing starting on [page 511](#), implements a simple object inspector. For any given object, the inspector prints the value of all the object's attributes. If any attribute is itself an object, the class recursively inspects that attribute by treating it as a root object. Loops in the object graph are avoided in this simple algorithm. To simplify the example, the `LogMgr` class is used to display attributes.

The `ObjectInspector` class defines the public method **Inspect** to inspect each attribute and to print it in the following format:

*name = [ declared-type ] value*

For example:

#### Code Example 14-9 Using the `ObjectInspector` class

```
Num = [integer] 1234
Name = [TextData] Bill
```

Non-nil objects are printed recursively, indented by spaces; for example:

```
Name = [string] Bob
Address = [AddressType]
  Street = [string] 811 Pine Lane
  City = [string] San Jose
  State = [string] CA
AccountID = [TextData] ng10234
Info = [TextData] NIL
```

Arrays are printed as a series of numbered objects, for example:

```
Name = [string] Bill
Aliases = [array of TextData]
  [1] = [TextData] BillG1
  [2] = [TextData] Bill@acme.com
  [3] = [TextData] Bill.Smith
```

Or, if the elements are composite, as in the following example:

```
AddressBook = [array of AddressType]
  [1] = [AddressType]
    Street = [string] 811 Pine Lane
    City = [string] San Jose
    State = [string] CA
    Zip = [integer] 94303
  [2] = NIL
```

Virtual attributes and unsupported types are simply tagged and not inspected. Objects that are seen multiple times in a single inspection are tagged with “...” after the first occurrence.

The following code listing shows the implementation of ObjectInspector:

```
class ObjectInspector inherits Object
  has public
    Inspect(obj : Object, level : integer = 0);
    Init();

  has private
    Cache : Array of object; // Avoid loops in object graph
    log : LogMgr;
end;

method ObjectInspector.Init()
begin
  self.Cache = new;
  log = task.Part.LogMgr;
end;

method ObjectInspector.Inspect(obj : Object, level : integer = 0)
begin
  // If passed object is NIL, return
```

```

if (obj = NIL) then
  log.Putline('NIL');
  return;
end;

//If passed obj is a DataValue, print its text form
if (obj.IsA(DataValue)) then
  log.Putline(DataValue(obj).TextValue);
  return;
end;

//Avoid printing-loops in object graph (tag output with ...)
if (self.Cache.FindRowForObject(obj) != 0) then
  log.PutLine('...');
  return;
end;

self.Cache.AppendRow(obj);

//Set up indent level, which increases with recursive calls
log.Putline();
indent : Textdata = new;
for i in 1 to level do
  indent.Concat(' ');
end;

//If passed object is array, print as [index] = [type] value
if (obj.IsA(GenericArray)) then
  arr : GenericArray = GenericArray(obj);
  for i in 1 to arr.Items do
    log.Put(indent);
    log.Put('[');
    log.Put(i);
    log.Put('] = ');
    elem : Object = arr.FindObjectForRow(i);
    if (elem != NIL) then
      log.Put('[');
      log.Put(elem.GetClassType().GetName());
      log.Put('] ');
    end;
    self.Inspect(elem, level + 1); //recursively inspect elem
  end;
  return;
end;

//print each attribute of passed obj as name = [type] value
cls: ClassType = obj.GetClassType();
ats : array of AttributeDesc = cls.GetAttributes();
for a in ats do
  log.Put(indent); //Name & type correctly indented
  log.Put(a.GetName());
  log.Put(' = [');

  if (a.IsVirtual()) then //If virtual just tag output
    log.Putline('virtual]');
    continue;
  end;
end;

```



```

    end;

    typ : ClassType = a.GetType();
    log.Put(typ.GetName());
    log.Put('] ');

    // If the type is primitive, confirm it's supported
    if (typ.IsPrimitive()) then
        if (not PrimitiveDesc(typ).IsSupported()) then
            log.Putline('unsupported type');
            continue;
        end;
    end;

    //recursively inspect actual value. (GetValue returns
    //a DataValue for supported primitive types).
    self.Inspect(a.GetValue(obj), level + 1);
end;

//Clear inspector cache at end of root object.
if (level = 0) then
    self.Cache.Clear();
end;
end;

```

## Class Browser

The `ClassBrowser` class, shown in the next listing, implements a simple class browser. For any given class, the browser prints the attributes and methods of that class, and then continues with its super class until it reaches the root `Framework.Object` class.

The `ClassBrowser` class has one public method, `Browse`, which analyzes each class and prints information about the class in the following format:

```

class-name:
    visibility attribute-name : { virtual | type }; ...
    visibility method-name([parameters]) : return-type; ...
super-class-name:
    ...

```

Each parameter is formatted as:

```

mechanism-name : type [,]

```

Unsupported types are tagged with **unsupported**.

For example, `UserAddress`, which subclasses from `Address`, would look like this:

**Code Example 14-10** Using the `ClassBrowser` class

```

UserAddress:
    private Link : virtual;
    private Aliases : Array of TextData;
    public AddAlias(input alias : TextData) : boolean;
Address:
    public Street : TextData
    public City : TextData;
    public State : TextData
    public Zip : IntegerData;
    public Phone : PhoneEntry;
    private Key : integer;
    public GetKey() : integer;

```

Here is the implementation of `ClassBrowser`:

```

class ClassBrowser inherits Object
    has public
        Browse(cls : ClassType);

    has private
        BrowseName(m : MemberDesc);
        BrowseType(typ : ClassType);
        BrowseParameter (p : ParameterDesc);
        log : LogMgr;
end;

method ClassBrowser.Browse(cls : ClassType)
begin
    log = task.Part.LogMgr;
    log.Putline();
    while (cls != NIL) and (cls != Framework.Object) do
        log.Put(cls.GetName());
        log.Putline(':');

        for a in cls.GetDeclaredAttributes() do
            self.BrowseName(a);
            log.Put(' : ');
            if (a.IsVirtual()) then
                log.Put('virtual');
            end;
            self.BrowseType(a.GetType());
            log.Putline(';');
        end;

        for m in cls.GetDeclaredMethods() do

```

```

        self.BrowseName(m);
        log.Put(' ');
        first : boolean = true;
        for p in m.Getparameters() do
            if (not first) then
                log.Put(', ');
            end;
            first = false;
            self.BrowseParameter(p);
        end;
        log.Put(' ');
        typ : ClassType = m.GetReturnType();
        if (typ != NIL) then
            log.Put(' : ');
            self.BrowseType(typ);
        end;
        log.Putline(';')
    end;

    cls = cls.GetSuperClass();
end;
log.Putline();
end;

method ClassBrowser.BrowseName(m : MemberDesc)
begin
    log.Put(' ');
    if (m.IsPublic()) then
        log.Put('public ');
    elseif (m.IsProtected()) then
        log.Put('protected ');
    else
        log.Put('private ');
    end;
    log.Put(m.GetName());
end;

method ClassBrowser.BrowseType(typ : ClassType)
begin
    if (typ = NIL) then
        return;
    end;
    if (typ.IsPrimitive()) then
        pd : Primitive Desc = PrimitiveDesc(typ);
        if (not pd.IsSupported()) then
            log.Put('[unsupported] ');
        end;
        log.Put(typ.GetName());
    elseif (typ.IsArray()) then
        ad : ArrayDesc = ArrayDesc(typ);
        t : ClassType = ad.GetArrayType();
        log.Put(t.GetName());
        t = ad.GetElementType();
        if (t != NIL) then
            log.Put(' of ');
            self.BrowseType(t);
        end;
    end;
end;

```

```
        end;
    else
        log.Put(typ.GetName());
    end;
end;

method ClassBrowser.BrowseParameter(p : ParameterDesc)
begin
    case (p.GetMechanism())
    when LO_PARAM_INPUT do
        log.Put('input');
    when LO_PARAM_OUTPUT do
        log.Put('output');
    when LO_PARAM_INPUT_OUTPUT do
        log.Put('input output');
    when LO_PARAM_CP_INPUT do
        log.Put('copy input');
    when LO_PARAM_CP_OUTPUT do
        log.Put('copy output');
    when LO_PARAM_CP_INPUT_OUTPUT do
        log.Put('copy input output');
    end;
    log.Put(' ');
    log.Put(p.GetName());
    log.Put(' : ');
    self.BrowseType(p.GetType());
end;
```

# XSLT Processor Library

This chapter provides a conceptual overview of the XSLT Processor library. The iPlanet Integration Server (iIS) Backbone product provides a built-in processor for performing such transformations between backbone-integrated applications. If you are not using iIS, you can use the TOOL XSLT Processor library to transform data exchanged between any TOOL applications.

For detailed descriptions of the classes in the XSLT Processor Library, see the iIS online Help.

For more information about iIS, see the website <http://sun.com/forte/fusion>.

## Features of the iIS XSLT Processor

An XSLT processor that you utilize with the iIS XSLT Library has the following key features and capabilities:

- It is compatible with other XSLT processors, including the one provided with the iIS product.
- It accepts XML files as input and provides XML or HTML 4.0 as output.
- It provides URL protocol handling, using the file protocol as the default and allowing you to specify other standard protocols or custom protocols.

For information about protocol handling, see [“Using Protocol Handlers” on page 523](#).

### Restrictions

The XSLT library does not provide the following features:

- You cannot specify output code-sets in your stylesheets.

- The XML parser is non-validating, that is, it checks that the input document is well-formed XML, but it cannot validate the document against a Document Type Definition (DTD). Therefore the Xpath Id() function does not work.
- The processor does not implement the <xsl:key> element, <xsl:fallback> element, or backwards compatible processing.

## Introducing XML and XSL

This section provides an introduction to:

- the Extensible Markup Language (XML) that you can use to transfer data between applications
- Extensible Stylesheet Transformations (XSLT) that you use to translate XML data between applications
- XSLT Processors that perform such transformations

## What is XML?

Extensible Markup Language (XML) is a method for structuring data in an application, generally for the purpose of sending the data to another application. XML lets you represent the data in a tree structure that depicts the relationships between the various elements of the data. For example, an application might generate an XML document like the following to represent a customer record:

**Code Example 15-1** A typical XML document generated by an application

```
<Customer>
  <CustName>
    <FirstName>John</FirstName>
    <LastName>Doe</LastName>
  </CustName>
  <CustAddress>
    <Street>123 Main Street</Street>
    <City>Island City</City>
    <State>CA</City>
    <Zip>95050</Zip>
  </CustAddress>
  <CustNumber>
    1234567
  </CustNumber>
</Customer>
```

In an XML document, the items between brackets are elements that describe the data, and the items between the elements are the data itself. XML looks similar to HyperText Markup Language (HTML), except that XML represents the content and structure of the data, rather than its presentation.

While you can assign arbitrary meaning to the elements in XML documents, the documents themselves cannot be processed unless they conform to a prescribed syntactical structure in order to be processed. For example, tags representing the beginning of an element must have a subsequent tag to close the element. A document that meets the XML syntax requirements, regardless of its content, is called a *well-formed* document.

For the XML specification and related information, see <http://www.w3.org/XML/>.

## Representing an XML Document in a TOOL Application

The XML source document that your TOOL application provides to the XSLT Processor for transformation can be in either of the following forms:

- An object of the `SeekStream` class, an abstract class representing stream objects that iIS can access for read and write operations. The three subclasses of the `SeekStream` class represent the possible sources of the data:
  - `File` represents an operating system file
  - `MemoryStream` represents text streams contained in memory
  - `BufferedStream` represents text stored in a buffer

For more information about `SeekStream` objects, see the iIS online Help.

- A Uniform Resource Locator (URL) corresponding to a text file containing the source XML data. The file must contain a well-formed XML document.

To represent the URL of the file, instantiate an object of the `URL` class and initialize it appropriately.

## What are XSL Transformations?

Applications can send data in the form of XML documents to other applications for various purposes. For example, you might want to send the customer record document to a credit verification application.

Because XML has no restrictions on which elements a document contains, however, the two applications might not have the same XML vocabulary and structure. As a simple example, the customer number might be represented as <CustNumber> to the first application, and as <CustID> to the second.

For this reason, it is often necessary to transform an XML document when sending it to another application. The mechanism for achieving this task is called Extensible Stylesheet Transformations (XSLT).

The other major use of XSL transformations is to produce HTML pages.

XSLT applies a *stylesheet* to a source XML document to render it as a result tree, which is output as XML, HTML, or plain text in the results document. The second application can then use the data in the results document as needed.

A stylesheet is itself an XML document that transforms XML by means of pattern matching. That is, when the stylesheet finds a specific pattern in the source document, it applies a template that specifies how to render it in the results document.

For example, the following stylesheet template matches the <CustNumber> element in the source document and transforms it into the <CustID> element in the results document:

**Code Example 15-2** Using XSLT to transform a document

```
<xsl:template match="CustNumber"> //Match CustNumber element
<xsl:element name="CustID"/> //Create CustID element
</xsl:template> //End of template
```

In this example, the “/” near the end of the second line serves as the close tag to ensure that the stylesheet (which is an XML document itself) is well-formed.

XSLT provides a wide variety of transformations that let you match complex patterns and perform sophisticated transformations. Your stylesheet can not only change the structure of the XML, but perform calculations and other changes upon the data itself.

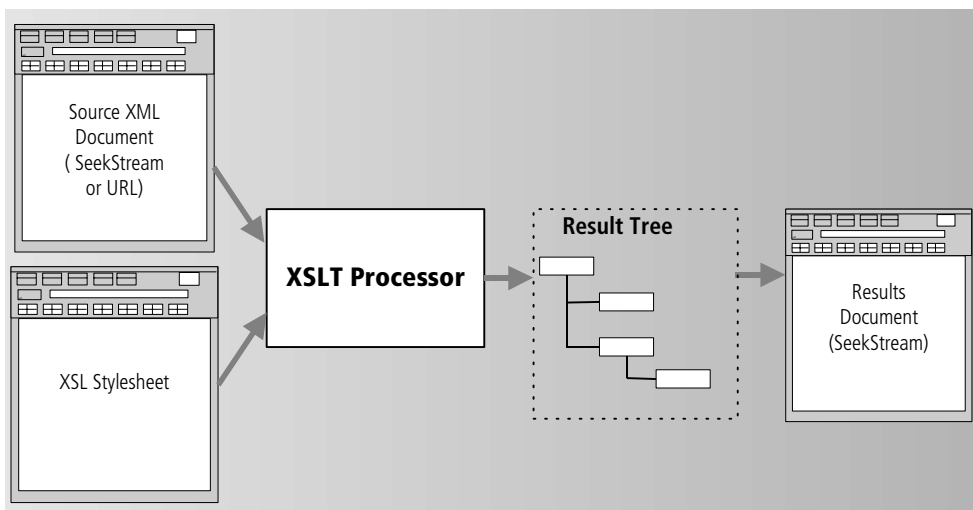
For the complete description of available XSLT elements for performing transformations, see the XSL specification at <http://www.w3.org/TR/XSLT>. If you are an iPlanet Integration Server customer, see the *iIS Backbone Integration Guide*.



## What is an XSLT Processor?

An XSLT Processor is the software tool that applies a stylesheet to a source XML document to render an XML results document. **Figure 15-1** illustrates this process.

**Figure 15-1** XSLT Processor Document Flow



The XSLT processor functions as follows:

- It transforms the source document into a tree structure.
- The processor steps through the source tree looking for a matching template in the stylesheet, beginning with the source root.
- If it finds a matching template, it applies its pattern to render a new entity in the results document.
- The processor continues through the source document until it has processed all the nodes, using supplied and default templates to direct further pattern matches.

Because the processor starts from the source document, your stylesheet can contain templates that are never applied, because the source document has no node that matches them. You can take advantage of this processing model to write a single stylesheet that you can apply against a variety of different source documents.

# Using an XSLT Processor in a TOOL Application

A TOOL application might typically use an XSLT processor as follows:

1. Create a processor instance.
2. Initialize the processor instance.
3. If you are accessing a source document through a protocol other than the file protocol, create a protocol handler for the protocol you are using.  
For more information about protocol handlers, see [“Using Protocol Handlers” on page 523](#).
4. Select a stylesheet.
5. Optionally precompile the stylesheet.
6. Call the appropriate version of the Parse() method on the processor instance to parse the XML input document:
  - o If Parse() returns TRUE, it will have created an XML results document for use as needed.
  - o If Parse() returns FALSE, call the processor’s GetErrorContext() method to determine why the input document could not be processed.

The following code sample illustrates the above steps:

## Code Example 15-3 Using the XSLT processor in an application

```
processor:XSLTProcessor=new();
sourceURL:URL=URL().Initialize('file:///alpha.xml');
sheetURL:URL=URL().Initialize('file:///omega.xsl');
result:File=new();
success:boolean;

result.SetPortableName('result.xml');

processor.SetSheetSource(sheetURL);
success=processor.Parse(sourceURL, result);

if success then
  // carry on...
else
  //Report error.
  context:ErrorContext=processor.GetErrorContext();
  task.lgr.putline('Transformation failed:');
  task.lgr.putline(context.GetMessage());
  task.lgr.put('  line:');
```

**Code Example 15-3** Using the XSLT processor in an application (*Continued*)

```
task.lgr.putline(context.GetLineNumber());
task.lgr.put(' in sheet:');
task.lgr.putline(BooleanData(value=context.IsInSheet()));
task.lgr.put(' in source:');
task.lgr.putline(BooleanData(value=context.IsInSource()));
end if;
```

## Using the Results Document in a TOOL Application

Although it is not required, in most applications you should ensure that your stylesheet is written to produce a well-formed XML document or HTML pages as its output. This will allow other components to read the results of the transformation.

The most common way to use the results document is to produce a `MemoryStream` object as the result of the transformation by the XSLT processor. You then can use that object to create an `InputSource` object for the XML parser.

For more information about the `XMLParser` library, see the iIS online Help.

## Using Protocol Handlers

A URL consists of two parts:

- a protocol, indicated by the part before the colon (:), specifying how to access the URL. This is sometimes referred to as a scheme.
- a resource, indicated by the part after the colon (:), specifying a location on the network

For example, in the URL `http://www.mysite.com/index.html`:

- `http` indicates the Hypertext Transport Protocol
- `www.mysite.com` is the host computer to be accessed
- `index.html` is the filename to be accessed from the host computer.

You might need to use any of several communication protocols to access the source XML document to be processed by your XSLT processor. By default, the processor expects the protocol to be “file,” that is, the location of a local source file. For example:

```
file:///c:/mydrive.my.xml
```

Besides file, standard protocols include http, ftp, gopher, email, and news.

There are also application-specific protocols. To specify any protocol other than file to use to access the source document, you must implement a protocol handler for your XSLT processor. The purpose of a protocol handler is to implement a protocol to access an XML source document whose URL includes the custom protocol. For example, in the URL `forte://xmldocuments.com`, forte is a custom protocol for which you would need to write a protocol handler. The same is true for any protocol other than file.

To implement a custom protocol handler, you must create two subclasses: one of `URLConnection`, the other of `URLStreamHandler`.

The `URLStreamHandler` class is responsible for creating connections to resources and, optionally, parsing the URL specification string. The `URLConnection` class is responsible for creating a `SeekStream` to the resources specified by a URL.

The custom protocol handler class methods is automatically called by the XSLT processor as needed; no further application action is needed after you register the protocol handler.

The following code example shows how to implement a custom protocol handler:

**Code Example 15-4** Implementing a custom protocol handler

```

method MyProtocolStreamHandler.OpenConnection
(input url: XSLT.URL): XSLT.URLConnection
begin
return MyProtocolURLConnection().Initialize(url);
end method;

method MyProtocolURLConnection.Connect
begin
if not GetConnected() then
url:URL = GetURL();
self.Stream = new();
self.Stream.Open(SP_AM_WRITE);
self.Stream.WriteLine('<?xml version="1.0"?>');

if url.GetFile()='/alpha.xml' then
self.Stream.WriteLine('<greeting>Guten Tag</greeting>');
elseif url.GetFile()='/omega.xsl' then
// etc
end if;

self.Stream.Close();
self.Stream.Open(SP_AM_READ);
self.Stream.Offset=0;
SetConnected(TRUE);
end if;
end method;

method MyProtocolURLConnection.GetInputStream:
Framework.SeekStream
begin
Connect(); --will connect only once.
return self.Stream;
end method;

```

The following code sample shows how to register a new protocol handler:

```

success:Boolean;
success=URLHelper().AddProtocol(
'forte',
MyProtocolStreamHandler());

```

# The XSLT Processor Library Classes

The XSLT Processor library provides classes used to transform XML documents in iIS programs in conformance with the W3C public recommendation on XSLT 1.0.

The XSLT Processor library contains the following classes:

Class	Description
XSLTProcessor	Creates an XSLT processor and performs XSL transformations.
ErrorContext	Conveys error information from the processor
URL	Represents a Uniform Resource Locator that specifies the location of an XML input document and the communications protocol (such as HTTP or FTP) for accessing the document
URLConnection	An abstract base class that lets you create a URLConnection object that you can use to communicate with the URL for reading the source document
URLStreamHandler	An abstract base class that lets you locate a source document by parsing a URL string based on the protocol and specified start and end points within the string
URLHelper	A utility class to register handlers for communications protocols other than the basic ones (such as http, ftp, gopher, or news) provided by the XSLTProcessor Library

All these classes inherit from Framework.Object. These classes are in the XSLT system library; you must use XSLT as a supplier plan to use these classes.

For detailed information about the attributes, methods, and syntax of each XSLT Processor Library class, see the iIS online Help.

Also, the following classes are modeled after their Java equivalents:

- URL
- URL Connection
- URL StreamHandler

For detailed descriptions of the Java classes and their usage, see the Java documentation at <http://java.sun.com/docs/books/tutorial/networking/urls>.

# Source Code Management for iPlanet UDS Projects

This chapter introduces the `GenericRepository Library` and the `SourceCodeManager Library`.

The `GenericRepository Library` in the iPlanet Unified Development Server runtime includes classes that allow the creation of a bridge between a central repository and a source code management system. This bridge is written in TOOL and is called a source code management service (SCM service).

The `SourceCodeManager library` in the UDS runtime implements a facility for storing multiple versions of a file in a single archive file.

You can implement a source code management (SCM) system in TOOL using the UDS `SourceCodeManager library`. You can also use the SCM service to connect to an external SCM system, either by a TOOL wrapping of the SCM system's callable interface or by using the `OperatingSystem.RunCommand` method to access the SCM system's command-line interface.

For detailed descriptions of the classes in the `GenericRepository Library` and the `SourceCodeManager Library`, see the UDS online Help.

The chapter contains the following sections:

- [“Overview of Source Code Management” on page 528](#)
- [“Source Code Management Service” on page 528](#)
- [“Using the SourceCodeManager Library” on page 531](#)
- [“Export Formats” on page 533](#)

# Overview of Source Code Management

When you (as a developer) integrate a workspace into the central repository, all changes in the workspace become part of the repository's system baseline. This allows other developers to see the changes when they update their workspaces.

If the central repository is connected to a source code manager service (SCM service), the changes are also exported as text, allowing the changes to be checked into a source code management (SCM) system.

## Source Code Management Service

UDS provides a way to create a bridge between a central repository and a source code management system. Using the `GenericRepository` Library, you define the interaction between the central repository and a source code manager. This source code manager can be a third-party product or one created using the `SourceCodeManager` Library.

## Features and Limitations

The `GenericRepository` Library contains classes that allow you to create an SCM service. These classes allow you to do the following:

- describe the plans and components that are being integrated
- control which plans and components to export
- give the SCM service control after the files are exported, so the SCM service can then check them into an external source code manager

The interaction between UDS and a source code manager consists of your implementation of the `BeforeIntegration`, `BeforeExport`, and `AfterExport` methods of the `SCMServer` class inside the `GenericRepository` library. You must implement them in a way that works with the external source code manager you are using.



## Using the GenericRepository Library

Inside the GenericRepository Library are four classes that implement interaction between a UDS repository client and an external source code manager.

The GenericRepository Library provides the classes listed in [Table 16-1](#).

**Table 16-1** Classes in the GenericRepository Library

Class	Description
SCMInfo	Contains attributes used to describe the user, repository, and export mode for an entire workspace integration.  The ExportMode attribute in this class defines the file export style described in <a href="#">“Export Formats” on page 533</a> . You can set the ExportMode to <code>SCM_EXPORT_MODE_COMPAT</code> , or <code>SCM_EXPORT_MODE_MULTIPLE</code> .
SCMObject	Contains attributes used to describe a single plan or component integrated as part of the current workspace integration.
SCMPlan	Contains attributes used to describe a plan and its components being integrated as part of the current workspace integration.
SCMServer	Defines the methods called by the repository client during an integration. These methods interact with a source code manager.

The UDS online Help provides full method and attribute descriptions for each of the classes in the GenericRepository Library.

### Using SCMServer

The SCMServer class contains the methods that build the bridge between the central repository and the source code management system. The SCMServer class contains three methods: `BeforeIntegration`, `BeforeExport`, and `AfterExport`. These methods are called by the repository client code during a workspace integration.

The `GenericRepository.SCMServer` class provides the methods listed in [Table 16-2](#).

**Table 16-2** Methods in the `GenericRepository.SCMServer` class

Method	Description
<code>BeforeIntegration</code>	The repository client calls this method before the integration is started. During this method you can collect information that is to be used later or verify that the integration should be allowed. The repository's global lock is held exclusively while <code>BeforeIntegration</code> runs, preventing any other integration or updates from occurring. Because of this, it is a good idea to minimize the amount of time <code>BeforeIntegration</code> takes to run.
<code>BeforeExport</code>	The repository client calls this method after the integration has finished successfully, but before any plans or components have been exported. This method lets you specify which plans and components to export and to which directories and files. The repository's global lock is held shared while <code>BeforeExport</code> runs, allowing updates but not other integrations.
<code>AfterExport</code>	The repository client calls this method after the plans and components marked by <code>BeforeExport</code> have been exported. During this method you can process the export files by checking them into the SCM system. It is also called if errors have occurred previously during the integration, so you can perform any needed cleanup. The repository's global lock is held shared while <code>AfterExport</code> runs, allowing updates but not other integrations.

The UDS online help contains sample code for each of the methods in the `GenericRepository.SCMServer` class.

# Using the SourceCodeManager Library

The SourceCodeManager library in the iPlanet UDS runtime implements a facility for storing multiple versions of a file in a single archive file.

## Features and Limitations

The archives produced by a SourceCodeManager service object are RCS-compatible. This allows RCS to examine the archives and add versions to them. These versions must always be added as a new head revision. For instance: you can add a version 1.3 to an archive whose current head is 1.2.

If RCS is not available, then the SCM utility, which is supplied with UDS as a command-line utility, can be used. For help on using the scm utility, type `scm help` to the command-line.

The SourceCodeManager.SCM class provides the methods listed in [Table 16-3](#):

**Table 16-3** Methods in the SourceCodeManager.SCM class

Method	Description
Startup	Set the directory that contains the archives.
CreateVersion	Check a new version into an archive.
GetVersion	Create a file from a version in an archive.
DeleteArchive	Delete an archive.
Differences	Display differences between two files.
Merge	Merge the differences between two files and a third file.

## Creating an SCM Service

Use the GenericRepository library to create a service object that provides source code management. The class GenericRepository.SCMServer defines methods that are called during an integration to provide the SCM service with files it can check into an SCM.

► **To create an SCM service**

1. Create a project in your workspace.
2. Create a new non-window class in the project:
  - Specify GenericRepository.SCMServer for the superclass.
  - Under Runtime properties specify distributed, shared object.
3. Implement the following methods that will be called during integration:
  - BeforeIntegration
  - BeforeExport
  - AfterExport

For examples of these methods, see the UDS online help for GenericRepository.SCMServer.

4. Create a new service object based on the type of this new class.
5. Configure the project as a server partition, then distribute and install the project.
6. Start up the central repository server that uses the SCM service, using the following flag:

**-scm** *source\_code\_manager*

*source\_code\_manager* is in the format *appname/projname/svcobjectname*.

## SCM Service Example

Your UDS distribution includes an example of an SCM service. The example uses the SourceCodeManager library to create archive files.

The example is in the following directory:

```
FORTE_ROOT/install/examples/scm/sourcecodemanager/scmserver.pex
```

To run the example, create an environment variable named `ROOT_DIRECTORY`, which is the root directory where TOOL source code archives will be written.

Import the `scmserver.pex` file into a central repository and go to [Step 5](#) of the above procedure.

# Export Formats

UDS Source Code Management provides two export formats for TOOL projects. One format is for compatibility with the SCM Hooks provided in older versions of UDS. The other is recommended for all new SCM Services. The SCM Service chooses which format to use by setting the `ExportMode` attribute of the `SCMInfo` object to either `SCM_EXPORT_MODE_COMPAT` or `SCM_EXPORT_MODE_MULTIPLE`.

In all cases, plans other than TOOL projects (for example, Application Models) are exported as a single `.pex` file.

## Compatibility Format

In this format, each changed TOOL project can be exported as a `.pex` file, and each changed TOOL component can be exported as a `.cex` file. This format is used when the `ExportMode` attribute of the `SCMInfo` object is set to `SCM_EXPORT_MODE_COMPAT`.

**Pex files** The `.pex` files can be imported into a repository to re-create the state of the workspace when the integration was done. However, because these `.pex` files can be large and contain many components, interpreting the differences between two versions of a `.pex` file can be difficult.

**Cex files** The `.cex` files are useful to track the history of the development of a single component, but do not by themselves contain the information needed to re-create the state of the workspace.

## Multi-File Format

In this format, UDS exports TOOL projects and components as multiple files. The number and type of files depends upon the type of the component. This format is used when the ExportMode attribute of the SCMInfo object is set to SCM\_EXPORT\_MODE\_MULTIPLE.

Table 16-4 describes the types of files that can be exported in multi-file format.

**Table 16-4** Types of Files Exported in Multi-file Format

Type of File	Description
Project level definition file <i>projectname.prx</i>	Contains project-level information, such as the project-level constants and the list of supplier projects.  It also contains a #include statement for each of the component-level files in the project. Thus, importing this file into a repository imports the entire project.
Component definition file <i>componentname.cdf</i>	A component definition file is exported for each class, cursor, and service object in the project. It contains all definitional parts of the component.
Class Implementation file <i>componentname.cex</i>	Exported only for classes, this file contains all of the method implementations for the class.
Class window definition file <i>componentname.fsw</i>	Exported only for window classes, this file contains the window definition for the class.

Table 16-5 lists the files, based on component type, exported in multi-file format mode.

**Table 16-5** Files Exported Based on Component Type

Component Type	Exported Files
Window classes	<i>componentname</i> .cdf <i>componentname</i> .cex <i>componentname</i> .fsw
non-Window and non-Interface classes	<i>componentname</i> .cdf <i>componentname</i> .cex
all other components, including Interfaces	<i>componentname</i> .cdf

The multi-file format allows convenient tracking of the history of any single component, and also allows the state of the workspace at the time of the integration to be re-created by importing the appropriate .prx files.

**NOTE** With the multi-file export format, the export filenames cannot be overridden by the SCM service; only the directory location can be controlled by the SCM service.

Also, because multiple projects could have like-named components, it is a good idea to export each project into its own unique directory to avoid name collision problems.





# Performance-Based Load Balancing

iPlanet UDS provides two different styles of load balancing for a service object:

- performance-based CPU allocation
- customized load balancing

This chapter discusses how to implement performance-based load balancing. Customized load balancing is the default, and is the same style of load balancing that is described in the Partition Workshop chapter of *A Guide to the iPlanet UDS Workshops*.

## Using Performance-Based Load Balancing

To use performance-based load balancing for a service object, you must:

- provide performance information for each node in the environment to which the service object will be assigned (usually performed by the system manager)
- turn on performance-based load balancing for the service object and specify a total number of replicates to be created
- for each configuration, indicate on which nodes the service object should run

The following sections provide detailed information about each of these tasks.

---

**CAUTION** Although you normally can run iPlanet UDS clients and servers from different releases within a single environment, you cannot use the performance-based load balancing feature unless both the client and the Environment Manager are of the same version (Release 3.0.F or greater). Do not try to use the performance-based load balancing feature if you have upgraded your client but not the Environment Manager.

---

## Providing Performance Information About a Node

Your system manager provides performance information for each node in the environment by using the Environment Console or Escript.

### Using the Environment Console

The Node Properties dialog in the Environment Console includes the following fields: Performance Rating and Number of Processors.

**Figure 17-1** Node Properties Dialog

The screenshot shows a dialog box titled "Node: MIMI". It contains the following fields and controls:

- Name:
- Architecture:
- Testing Node:
- Performance Rating:  Number of Processors:
- Client  Use as Model  Use for Testing
- Resource Managers:
- A table with two columns: "Name" and "Resource Manager". The table is currently empty.
- Buttons:

The Performance Rating property represents the performance rating for the specific node. The performance rating is an arbitrary number that represents the rating for an individual node on any scale that you choose. For example, nodes could be rated on a scale from 1 to 10 or from 1 to 100—the system manager can choose whatever seems appropriate.

If the system manager does not provide a performance rating for a given node, it is considered to be “average.” If the system manager does not provide performance ratings for any nodes, all are rated equal, and only the number of processors is used to balance the load.

The Number of Processors property is where the system manager simply enters the number of processors on the node. The default is 1.

## Using Escript

Escript provides the following Environment Edit Mode commands for specifying performance information about a node:

**SetNodeProcCount** *processor\_count*  
*performance\_rating*

These two commands are the same as the Environment Console node properties described above.

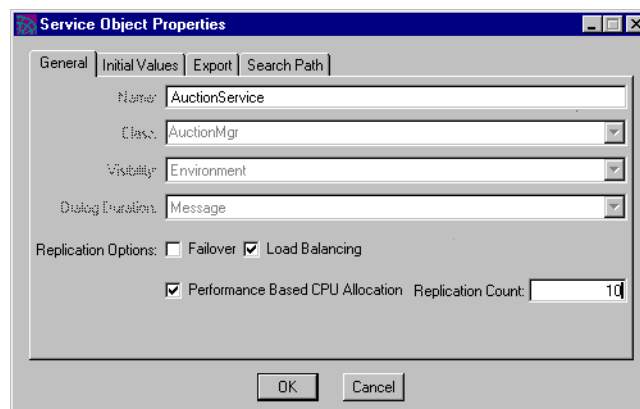
## Specifying the Number of Replicates

To use performance-based load balancing, you must explicitly turn on performance-based load balancing for the service object and then specify a fixed number of replicates for that service object. When you later partition the application in a given environment, iPlanet UDS ensures that the total number of replicates specified for the service object is evenly load balanced across the nodes to which it is assigned.

You can specify the total number of replicates for a service object in the Service Object properties dialog either in the Project Workshop or Partition Workshop. We recommend that you set the total number of replicates for the service object in the Partition Workshop because the total number of replicates typically varies for each environment.

The Service Object Properties dialog provides two fields for performance-based load balancing: Performance-Based CPU Allocation and Replication Count.

**Figure 17-2** Service Object Properties Dialog



The Performance-Based CPU Allocation property enables you to turn on performance-based load balancing. This toggle is enabled only after you have turned on the Load Balancing property in the same dialog.

If you do not turn on the Performance-Based CPU Allocation toggle, you get custom load balancing, which is described in the Partition Workshop chapter of *A Guide to the iPlanet UDS Workshops*.

If you do turn on the Performance-Based CPU Allocation toggle, iPlanet UDS provides performance-based load balancing for the service object based on the performance information that the system manager provided for the nodes in your environment and on the replication count you specify using the Replication Count field.

The Replication Count property specifies the total number of service objects to be assigned to nodes in the environment. After you assign the service object's partition to the nodes where you want the service object to run, iPlanet UDS automatically adjusts the replication count on each assigned partition so that the total number of replicates in the environment is equal to the number you specified in the Replication Count field.

By default, the Replication Count property is set to 1. If you set it to 0, iPlanet UDS turns off performance-based load balancing and reverts to custom load balancing. iPlanet UDS cannot provide performance-based load balancing with a replicate total of 0.

The TOOL **service object** statement has an option that allows you to enable performance-based load balancing for a service object within a .pex file:

**perfbasedalloc**=*integer*

The *integer* specifies the total number of replicates for the service object.

Remember, you must first turn on load balancing for the service object before you can enable performance-based load balancing.

For example:

```
so: MyClass = (loadbalance=TRUE, perfbasedalloc=30);
```

## Partitioning the Replicates

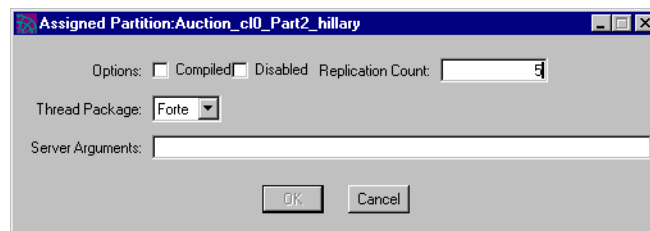
You assign the replicates to nodes the same way you normally do: drag the logical partition that contains the service object to each node where you want the service object to run. iPlanet UDS then automatically sets the replication count for the service object on that node so the total number of replicates on all the nodes to which it is assigned is equal to the service object's Replication Count property and so the load is evenly balanced across all the nodes.

Each time you assign the service object to a new node, iPlanet UDS automatically re-adjusts the replication counts on the rest of the nodes where the service object is already assigned.

Note that if you assign the logical partition to more nodes than the total number of replicates specified for the service object, iPlanet UDS assigns one replicate to each node, letting you exceed the total number originally specified for the service object. Also note that if two or more performance-based load-balanced service objects are combined in the same partition, iPlanet UDS uses the highest specified replication count of the two service objects to determine the total number of replicates for both service objects.

If you wish to check how many replicates are assigned to a given node, you can double-click the assigned partition to open the Assigned Partition Properties dialog. The Replication Count field displays the replication count for the service object on the partition.

**Figure 17-3** Assigned Partition Properties Dialog (Server Partition)



Note that when performance-based load balancing is enabled, the Replication Count property on the Assigned Partition Properties dialog is read only. You cannot change the replication count for an individual assigned service object because this is a number that iPlanet UDS sets automatically. However, you can always change the total used for the entire environment by changing the Replication Count field on the service object in the logical partition as described under [“Specifying the Number of Replicates” on page 539](#).

---

**NOTE** While the Replication Count field on the dialog for the assigned partition shows the total number of replicates assigned to the node, the Replication Count field on the dialog for the assigned service object shows the total number of service objects for the application (that is, the same number specified for the service object on the logical partition). For accurate information about the number of service objects on a given node, please use the Assigned Partition Properties dialog.

---

If the performance information about the nodes is changed in the Environment Console or Escript *after* you have partitioned the application, the new performance information will not be automatically reflected in the current partitioning scheme. To use the new performance information, you must repartition the application.

# Creating HTTP Applications

The HTTPSupport library allows you to create iPlanet UDS server and client applications that utilize the HTTP protocol. This chapter provides an overview of HTTP and application behavior. It then describes how to create iPlanet UDS client and server applications that use the HTTP and SSL protocols.

This chapter covers the following key topics:

- General HTTP concepts, including clients, servers, requests, responses, and sessions (“[HTTP Overview](#)” on page 543)
- How to create iPlanet UDS HTTP client applications (“[Creating HTTP Clients](#)” on page 553)
- How to create iPlanet UDS HTTP server applications (“[HTTP Servers](#)” on page 559)
- How to configure iPlanet UDS HTTP client and server applications (“[Configuring HTTP Sessions](#)” on page 567)

## HTTP Overview

In applications with client-server architecture, clients are programs that make requests for some sort of processing and servers are programs that listen for requests, process the requests, and respond to the client.

In the HTTP messaging model this is the only kind of exchange that takes place between clients and servers.

## Clients

In iPlanet UDS applications virtually any program can act as an HTTP client. The defining characteristic of a client in the context of HTTP messaging is that it requests services from a server. (See [“Creating HTTP Clients” on page 553](#) for further information.)

## Servers

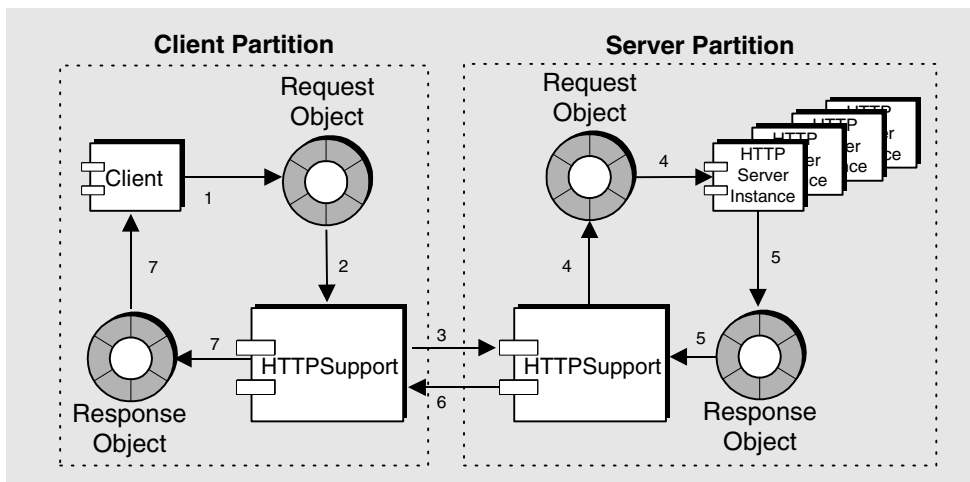
Any iPlanet UDS application that implements either the MessageReceiver or HTTPReceiver interface can act as an HTTP server. An HTTP server passively listens for requests from clients, providing request processing services. Users cannot interact directly with servers—all interaction with servers is mediated by the client program. Servers make themselves available to clients by listening at a specific network address. A client must know the address of a server before it can send requests to it. (See [“HTTP Servers” on page 559](#) for further information.)

## The HTTPSupport Model

The HTTP protocol uses a simple request/response model. In this model clients create request messages, which include headers, a body, and a session object. iPlanet UDS HTTP client applications can build a request and send it to a server instance. The server instance processes the request and sends back a response that includes headers with information about the processing and possibly a body containing data resulting from the processing.

**Figure 18-1** illustrates the HTTPSupport model by showing an iPlanet UDS HTTP client communicating with an iPlanet UDS HTTP server. In reality, an iPlanet UDS HTTP client can communicate with any HTTP server and an iPlanet UDS HTTP server can communicate with any HTTP client.



**Figure 18-1** HTTPSupport Model

1. The client creates a request.
2. The client sends the request to the client partition HTTPSupport.
3. The client partition HTTPSupport parses the request headers and body, translates the request message into a byte stream, and routes them to the server's network address using TCP/IP.
4. The server partition HTTPSupport receives the byte stream, creates from it a request object and pre-allocated response object, which it dispatches to the first available HTTP server instance.
5. The HTTP server processes the request and uses the pre-allocated response object to build the response message. The HTTP method `Process` is called, with the request object as its first parameter and the pre-allocated response object as its second parameter.
6. When the HTTP `Process` method terminates, control is passed back to the server partition HTTPSupport. The server partition parses the response object and renders it as a byte stream, which it then routes to the client using its network connection.
7. The client partition HTTPSupport receives the byte stream and creates from it a response object, which it routes back to the requesting client.

Brokering the exchanges between clients and server instances is HTTPSupport. HTTPSupport is responsible for managing sessions, parsing requests, dispatching requests to server instances, parsing responses, and forwarding them to the client that sent the request. HTTPSupport also manages system-wide configuration.

## Messages

The unit of communication between clients and servers is a message. HTTP messages consist of a header and a body. Typically, HTTPSupport automatically handles creating and reading HTTP headers and HTTP messages for you.

**Table 18-1** illustrates an example of a request message.

**Table 18-1** Typical HTTP request message

```
POST /my_domain.com/foo/bar/file.html HTTP/1.1
Date: Sat, 9 Dec 2000 17:41:22 GMT
Accept-Charset: ISO-646-DK;q=0.7, ISO-IR-158;q=0.4
Content-Length: 4078
Content-Type: text/html; charset =iso-8859-1

<DOCTYPE HTML PUBLIC "-//WC3//DTD HTML 4.0 Transitional//EN"
'http://www.w3.org/TR/Rec-html40/loose.dtd'>
<html>
...
</html>
```

In **Table 18-1**, the first line is a request line, describing the type of request. The next four lines are headers that pass information about the message from the client to the server. The headers are separated from the message body by a blank line.

Response messages are similar, except that the first line is a status line, indicating the success (or failure) of the request.

## Headers

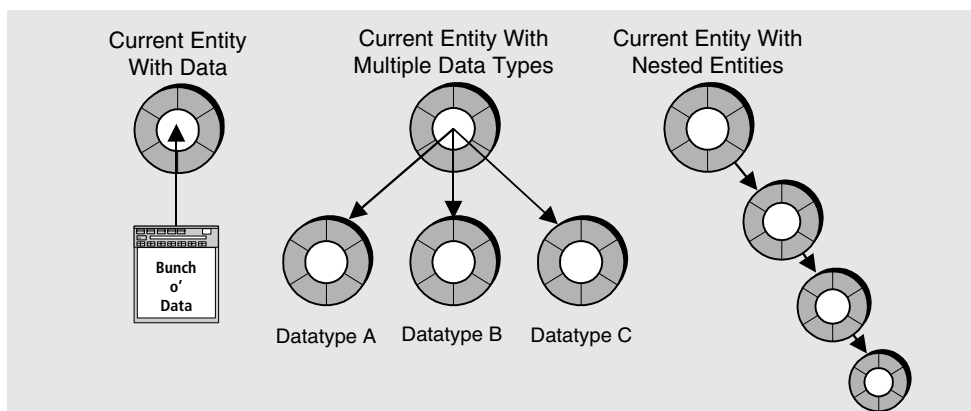
Headers consist of entries of the form *name:value*, where *name* is one of several classes of information defined in the HTTP specification and *value* is the actual value for that field. The order of the header entries is not mandated in the HTTP specification. However, by convention *general* headers appear first, followed by *request* headers, *response* headers, and *entity* headers.

- *General* headers specify information about the message itself. For example, a general header could date-stamp the message, or specify that the session should be closed once the server has responded to the request.
- *Request* headers are used by clients to specify the host making the request, acceptable response encodings, authorization data, and other information.
- *Response* headers are used by servers to include additional information about the response that cannot be contained in the status line. For example, if the response status line specifies “Method Not Allowed,” the response header would indicate which methods are supported for the resource specified in the request.
- *Entity* headers are used by clients to include meta-information about included entities or resources identified in the message, such as the type of encoding applied to the entity, or the natural language of the audience intended for the entity.

## Message Body

The message body is the actual contents of the message. By default the body contains an empty entity, which the client can either fill with data or replace with an entity of its own. The default entity has the media type *text/html*. The HTTP protocol defines a number of additional media types, and messages can contain any media type the recipient is capable of processing.

In addition to containing the default entity the message body can also contain an entity that contains other nested entities, or it can contain a number of separate entities. The diagram below illustrates these possibilities.



## Requests

Clients send *requests* to servers to ask for some type of processing. The client specifies the resource on which the processing will be done and the HTTP method to be used in the processing. A request can also contain data the client wants the server to process in some manner. (For more information see [“HTTP Client Requests” on page 553.](#))

## Responses

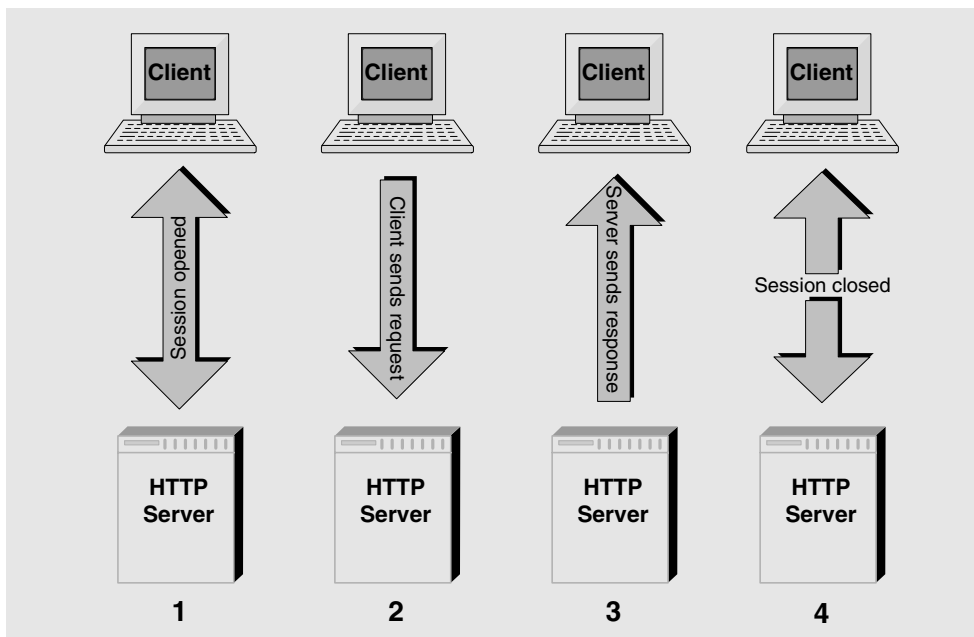
Responses enable the server to pass the results of the request back to the client. The results may be data of some type, an error code in the case of an invalid request, or a status code describing the outcome of the requested processing. Servers can also include cookies in a response, which are a means of maintaining session state information. (For more information see [“Listening For Requests” on page 561.](#))

## Sessions

For clients and servers to communicate with each other they need to first establish a *session*. A session is analogous to two people talking on the phone: before they can talk, one of them has to call the other, and the other has to pick up the phone when it rings. When the conversation is complete, they both hang up.

HTTP sessions can be established for a single exchange or can be kept open indefinitely in anticipation of multiple exchanges. HTTPSupport embodies three separate concepts of sessions: *application sessions*, *network sessions*, and *secure sessions*.

The following diagram illustrates an application session opened for a single exchange and then closed.



The following sections provide an overview of application, network, and secure session behavior.

## Application Sessions

An application session is an HTTP request/response exchange that is initiated by a client and established at the application level between the client and a server instance. Depending on how the server is configured, it may or may not grant the request for a session.

The HTTP protocol provides no means of maintaining state information during sessions. Once a server has received, processed, and responded to a request it no longer knows anything about that request/response exchange.

The server can “remember” the details of an exchange once the exchange has ended by including *cookies* with its response. Cookies are name/value pairs that enable the server to store session information with the client. The next time the client sends a request, it first checks to see if it has any cookies that are relevant to the current request URI, and if so it includes them in the request. When the server reads the cookies it remembers the state of the prior exchange.

One common example of how cookies are used is an e-commerce web site. The first time you visit such a site its server sends a number of cookies to your browser, which then stores them on your file system. The next time you point your browser (the HTTP client) to that site, it checks in its cookie directory to see if it has any cookies that apply to that URL. If it does it sends them to the site's web server, which reads the cookies, remembers you from the last time you visited that site, and re-establishes the context of your previous visit. This might include recognizing your login and associating it with a user preference, or remembering that you are interested in armadillos but not plaid pants.

## Network Sessions

A network session defines the behavior of the TCP/IP connection between the client and the server. A network session can be either held open for all messages that the client and server exchange, or open and close for each session. Establishing an open (persistent) TCP connection optimizes performance when the client anticipates having multiple exchanges with the server over a short period of time.

---

**NOTE** By default, HTTPSupport creates persistent network sessions. Not all HTTP 1.0 servers support persistent connections. Refer to [“Configuring HTTP Sessions” on page 567](#) for information on overriding default behavior.

---

Because opening and closing TCP connections is relatively expensive in terms of system resources it is more efficient to keep a single TCP connection open for multiple exchanges than to open and close a connection for each exchange. However, keeping a TCP connection open may block other clients from accessing the server. This occurs if there are more TCP connections open than the operating system configuration allows.

Network sessions have no knowledge of application sessions, and can be opened and closed independently of application sessions.

## Secure Sessions

Sites with applications handling proprietary or sensitive information can implement secure client-server communications with the Secure Sockets Layer (SSL) protocol. SSL provides three different types of security: *message secrecy*, *authentication*, and *message integrity*.

**Message Secrecy** Message secrecy is a means of ensuring that if a message is intercepted it cannot be read by the interceptor. This is implemented using various kinds of data encryption, such as DES or RC4.

**Authentication** Authentication is a means of ensuring that one or both communicating parties are who they claim to be. Many hacking techniques are based on fraudulently assuming the identity of an authorized user in order to gain access to privileged information. Authentication is implemented with public and private key certificates.

**Message Integrity** Message integrity is a means of ensuring that even if a message has been intercepted that its content has not been modified or re-sent. Message integrity is implemented with message digests.

For information on configuring secure sessions, refer to [“Configuring HTTP Sessions” on page 567](#). For additional information on the SSL protocol, refer to [Chapter 19, “Enabling Security” on page 581](#).

## HTTPSupport Classes and Interfaces

The TOOL classes that implement HTTPSupport, and their places in the class hierarchy, are shown in the following diagram.

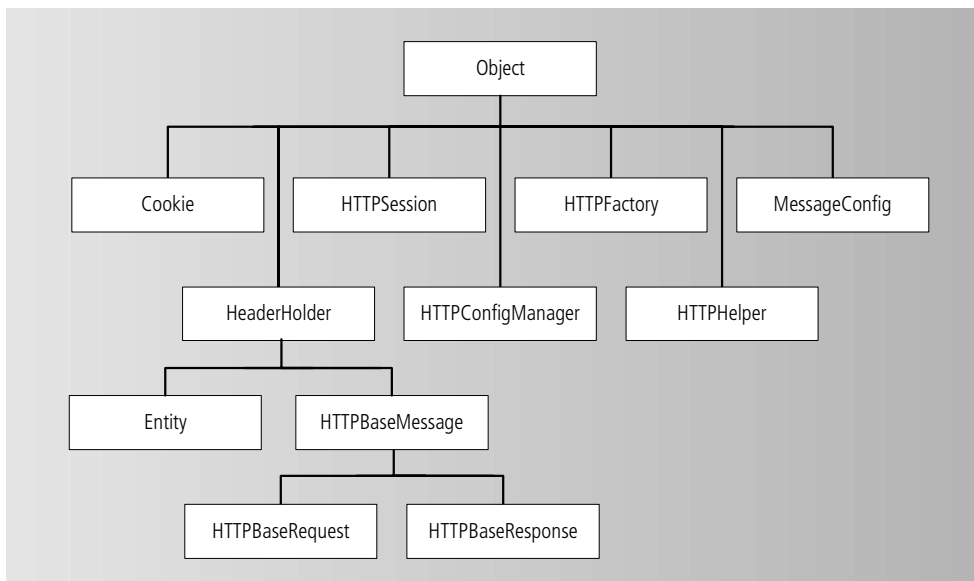


Table 18-2 summarizes each of the HTTPSupport classes.

**Table 18-2** HTTPSupport Classes

<b>Class</b>	<b>Description</b>
Cookie	Sets and gets cookie values.
Entity	Writes and reads data, enables nesting entities.
HeaderHolder	Sets, gets, adds, removes, and lists HTTP message headers and their values.
HTTPBaseMessage	Adds and gets cookies, sets and gets message values, encodes and decodes message bodies in Base64.
HTTPBaseRequest	Gets and sets URL, gets and sets HTTP methods, sets request URIs and query strings, sends requests
HTTPBaseResponse	Returns list of cookies, returns date header of message, sets and returns status code of response.
HTTPConfigManager	Sets and gets configuration values for TCP and SSL sessions.
HTTPFactory	Factory for MIME data containers and request, response, and session instances. Custom versions can be registered with HTTPFactory.
HTTPHelper	Advertises server, finds configuration manager and factory.
HTTPSession	Sets and retrieves information on network, application, and secure sessions.
MessageConfig	Utility class used for initializing message receivers.

HTTPSupport provides the following interfaces, many of which are implemented in the HTTPSupport classes:

```

GenericEntity
GenericHeaderHolder
GenericMessage
GenericRequest
GenericResponse
GenericSession
HTTPReceiver
HTTPServerManager
HTTPServerStateListener
MessageReceiver

```



# Creating HTTP Clients

Any iPlanet UDS applications can act as an HTTP client. Within your iPlanet UDS application, you create an HTTP message in a request object that is sent to an HTTP server. The client captures the response message from the server within a response object. The response object includes a session object that encapsulates information about the exchange between the client and server.

## HTTP Client Requests

An HTTP request message has two parts: message headers, which contain information about the message itself and how the client wants the message to be processed, and the body, which contains the data of the message.

Requests are constructed from the `HTTPBaseRequest` class. Requests must specify the method to invoke on the HTTP server and a string specifying the URL of the server. By default, the `HTTPSupport` library provides an empty body for your message and creates the HTTP headers to implement your request. You can optionally build a request that contains a query string, associate an entity or text with the message body, and set other HTTP message options for your request.

**Code Example 18-1** builds and sends an HTTP request.

### **Code Example 18-1** Building an HTTP client request

```
-- Create the request object
request : HTTPBaseRequest = new;

-- Specify the server HTTP method and the server's url
request.setMethod('GET');
request.setURL('http://iplanet.com:8080');

-- Send the request and capture the server's response
response : HTTPBaseResponse = request.send();

-- Place the response body in a TextData field
textField : TextData = new;
response.body.ReadText(textField);
```

## Specifying Request Details

This section highlights some of the additional specifications you can make when building an HTTP request. For complete documentation on building requests, refer to the online help for `HTTPBaseRequest`.

### SetMethod

`HTTPBaseRequest.SetMethod` method specifies the HTTP method used in the request. The syntax for `SetMethod` is:

```
SetMethod(string http_method)
```

*http\_method* specifies the HTTP method you want to invoke.

The possible values for *http\_method* are `OPTIONS`, `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `TRACE`, and `CONNECT`. If you do not specify this method, then by default, `HTTPBaseRequest.Send` specifies `GET`.

### SetURL

`HTTPBaseRequest.SetURL` method specifies the location of the server. The syntax of `SetURL` is:

```
SetURL(string url)
```

*url* is the fully specified absolute URL for the server.

The specified URL must include both the scheme and the path, and optionally the query. For example:

```
SetURL('http://my_domain.com/policies.shtml');
```

### SetQueryString

`HTTPBaseRequest.SetQueryString` method specifies query parameters that are used to build a URI. The query component of a URI passes small amounts of data to the resource addressed in the URI. A typical example is a search engine where the client passes search criteria to the engine as query parameters.

The syntax of `SetQueryString` is:

```
SetQueryString(string query)
```

*query* is the query string passed to the recipient.

For example:

```
SetQueryString('HTTP+tutorial');
```

results in a URI that might look like:

```
http://my_domain.com/search?q=HTTP+tutorial
```

## Message Body

`HTTPBaseMessage.SetBody` method specifies the contents of the body entity. By default, requests created by the client have an empty body of the `Entity` class. `SetBody` enables you to replace the default entity with one of your own. The syntax of the `SetBody` method is:

```
SetBody(GenericEntity body)
```

*body* is the entity you want to specify.

For more information on the `GenericEntity` Interface, refer to online help.

## Entities

The `Entity` class provides the default entity for containing data in HTTP messages. It provides methods for writing various kinds of data from the memory stream into an entity, setting the content type of the entity, and nesting entities. You can extend and customize the `Entity` class to accommodate different MIME types. For more information on using entities, refer to [“Entities” on page 565](#).

## Message Headers

HTTP message headers tell the server how the client wants the request processed, provide the server with information about the enclosed entities, and tell the server what kinds of responses it accepts. Typically, the `HTTPSupport` library automatically creates message headers for you. But you can use the following methods, available from `HTTPBaseRequest`, to create or add headers.

```
SetHeader(string name, string value)
SetIntHeader(string name, integer value)
AddHeaderValue(string name, string value)
```

`SetIntHeader` creates headers that take an integer value, (for example, `SetIntHeader('max-forwards', 5)`). `AddHeaderValue` creates multiple headers with the same name.

## Sending Messages

HTTP request/response exchanges can be either synchronous or asynchronous.

With *synchronous* processing (the default), the client must wait for a response to its request before doing anything else. The response of `HTTPBaseRequest.Send` is an `HTTPBaseResponse` object.

With *asynchronous* processing, the client can send a request and not be blocked as it waits for the response (it can continue with other processing).

To implement asynchronous processing, the client must specify a `MessageReceiver` object in the `HTTPBaseRequest.Send` method invocation. For asynchronously processed requests, the `Send` method immediately returns a `NULL` value so the client can continue with its normal processing. The server eventually returns the response object to the specified `MessageReceiver` in the request. The `MessageReceiver.Process` method is then automatically called with the request object as its first parameter and the response object as the second parameter.

### Send Method

There are three forms for the `HTTPBaseRequest.Send` method:

```
send([MessageReceiver signal])
send(string url, [MessageReceiver signal])
send(string methodName, string url, [MessageReceiver signal])
```

*signal* is an optional parameter that implements asynchronous messaging by specifying a `MessageReceiver` object.

*url* specifies the location of the server. If *url* is not specified, then the URL previously set with `SetURL` is used.

*methodName* specifies the HTTP method invoked in the request. If *methodName* is not specified, then `GET` is the HTTP method that is invoked.

### Dispatching Requests

HTTP servers can be implemented with multiple server instances. In the case of multiple server instances, a request goes to the first available server instance. If a client establishes an application session with a server instance, all further requests from that client within that session are sent to the same server instance.

If a request is sent and all server instances are busy, the request waits for the first available instance. Once a server instance receives a request it cannot receive another until it has finished processing the first request. If the client sends multiple, parallel requests to the same server instance they are processed serially in the order received.

## Sessions

When a client receives a response from the server, it creates a response object. The response object contains a session object that contains information about the interchange between the server and client. The session object contains configuration information about two different kinds of sessions: application session and network session.

---

**NOTE** Application sessions and network sessions are independent of each other. An application session can live longer than a network session and a network session can live longer than an application session. For more information, refer to *“Sessions” on page 548*.

---

You typically, do not need access to the session object—the HTTPSupport library handles details of the session for you. However, if you want to specify configuration information about the session or otherwise retrieve or modify information about the session, your request object must include a session object.

### *Application Session*

An application session defines the session between the client object and the server object. For example, if a client wants to send a series of requests to the same server instance, it uses an application session. The application session creates a virtual circuit between the two objects, which is held open until either the client or server closes the circuit. The default behavior for application sessions is that the client lets the server decide if an application session needs to be created.

### *Network Session*

A network session defines the behavior of the TCP/IP connection between the client and the server. The network session can be held open for all messages exchanged between the client and server, or open and close for each message. The default network session is persistent, which is held open for all messages exchanged. Not all HTTP 1.0 servers support persistent connections. Persistent connections is the default behavior for HTTP 1.1.

**Code Example 18-2** shows how to configure a network session for a request. In this example, the client is requesting HTTP 1.0 behavior for a network session—it specifies to close the TCP connection after each exchange.

**Code Example 18-2** Creating a session

```
-- Build the request
request : HTTPBaseRequest = new;
request.setMethod('GET');
request.setURL('http://iplanet.com:8443');

-- Create the session object for the request
session : HTTPSession = new;
request.SetSession(session);

-- Close the TCP session after each exchange
session.setSessionValue(HTTP_CONFIG_TCPSESSION_VALUE,
                        HTTP_TCPSESSION_MESSAGE)

-- Send the request
response : HTTPBaseResponse = request.send();
```

## Reusing Sessions

If an HTTP client application communicating with an HTTP server wants to use the same session when sending additional requests to the same server, the same session object must be reused. The following example shows how to use the `SetSession` method and the session attribute of a request to reuse a session object.

**Code Example 18-3** Reusing a session object

```
request : HTTPBaseRequest = new;
mySession : HTTPSession = new(applicationSession =
                             HTTP_SESSION_REQUIRED);
request.setSession(session);
request.send('http://engwww/index.html');

-- Processing the response
. . .

-- Reuse the same session
request2 : HTTPBaseRequest = new(session = mySession)
request2.send('http://engwww/pub/index.html');
```

# HTTP Servers

An HTTP server is any program that listens for HTTP requests from other programs and provides some sort of service in response. An HTTP server response message may include a body containing some sort of data requested by the HTTP client, or it may only have headers with information about how the request was processed or why it could not be processed. [Table 18-3](#) lists a typical response message from an HTTP server.

**Table 18-3** Typical HTTP response message from an HTTP server

```
HTTP/1.1 200 OK
Date: Sun, 3 Dec 2000 22:30:41 GMT
Last-Modified: Tue, 5 Nov 2000 04:17:56 GMT
Accept-Ranges: bytes
Content-Length: 4078
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//WC3//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/Rec-html40/loose.dtd">
<html>
... plain HTML text...
</html>
```

## Creating iPlanet UDS HTTP Server Applications

Any iPlanet UDS application that implements either the `MessageReceiver` or `HTTPReceiver` interface can act as an HTTP server. This enables the server to receive both the request and a pre-allocated response object.

## MessageReceiver Interface

An iPlanet UDS server application implementing the MessageReceiver interface must implement the following methods:

```
Initialize(MessageConfig config)
Process(GenericRequest request, GenericResponse response)
Terminate()
```

*config* is configuration information for the server session.

*request* is a pre-allocated request object generated from the client request.

*response* is a pre-allocated response object created to respond to the client request.

Initialize opens and configures a new session with a client. This method can be empty, in which case a session is opened with default settings. The Process method must implement code to handle incoming client requests. The Terminate method is called when the session is ended.

The MessageReceiver interface is more general than the HTTPReceiver interface. It is useful if you expect your client to specify only one or two HTTP methods that can be easily handled by the Process method.

## HTTPReceiver Interface

The HTTPReceiver interface extends the MessageReceiver interface, providing the following methods, which must be implemented by the iPlanet UDS server application:

```
DoDelete
DoGet
DoHead
DoOptions
DoPost
DoPut
DoTrace
```

The syntax of each of these HTTPReceiver methods is similar to the syntax for DoGet:

```
DoGet(HTTPBaseRequest request, HTTPBaseResponse response)
```

When a server that implements the HTTPReceiver interface receives an HTTP request from a client, the HTTPReceiver method that corresponds to the HTTP method specified by the client is automatically called.



For example, if the request specified the HTTP method GET, then the server application method `doGet` is automatically invoked.

---

**NOTE** In the situation where an unknown HTTP method is specified in a client request, `Process` is automatically called. You are responsible for writing code for handling such exceptions and returning the appropriate error code.

---

The `HTTPReceiver` interface is useful if the client requests any supported HTTP message, or if you want to create your own subclasses to specify behavior for handling specific HTTP methods.

## Listening For Requests

An iPlanet UDS HTTP server application can use the `Advertise` method from the `HTTPHelper` class to advertise the server. `Advertise` initializes and sets up the server to listen for HTTP requests. The `Advertise` call must be placed in an event loop so the server can listen for responses until the event loop closes.

Also, you can use the `SetServiceEOSInfo` command to enable an iPlanet UDS service object to listen for requests.

### Advertise Method

The syntax for `Advertise` is:

```
Advertise(object server, string 'option=value [...]')
```

*server* is the object you want to advertise as a server

*option=value* is an option name and a corresponding value. You can use multiple options, separated by white space.

**Code Example 18-4** creates a server that listens on port 8080, instantiates 12 instances of the server, and times out after four minutes.

#### **Code Example 18-4** Advertising a server

```
-- Instantiate a server
-- (myServer is a class that implements HTTPReceiver interface)
server : myServer = new;

-- create event loop to listen for requests
```

**Code Example 18-4** Advertising a server

```

event loop

-- Advertise the server
HTTPHelper().Advertise( myServer, 'port=8080, instances=12,
                           timeout=240' );

    when task.Shutdown do
        exit;
    end event;

```

For more information on configuring servers, including using the Advertise method to specify configuration information, refer to [“Configuring HTTP Servers” on page 570](#).

### SetServiceEOSInfo Command

If an iPlanet UDS service object implements either the MessageReceiver or HTTPReceiver interface, it can use the SetServiceEOSInfo FScript command to listen for HTTP requests. The syntax for this command is:

```
SetServiceEOSInfo ServiceObjectName httpdc port=nnnn
```

*ServiceObjectName* is the name of the service object.

*nnnn* is the port number used to listen for requests.

## Responding to Requests

When a client sends a request to an iPlanet UDS server application, the request is automatically routed to the corresponding HTTPReceiver method or the Process method. These methods are called with a pre-allocated request object as the first parameter and a pre-allocated response object as the second parameter.

The response object is derived from the HTTPBaseResponse class and is used by the server to construct its response. HTTPBaseResponse includes methods for reading request headers, accessing the body of the request, accessing the session object of the current session, setting response headers, and associating entities with the response.

## Sessions

An HTTP server never has to create a session object. The HTTPSupport library automatically creates a pre-allocated session object for each request received. This session object contains the values describing the nature of the session established between the server and client.

## Processing Cookies

Cookies are structured pieces of data stored on the client's disk to retain session state information for an HTTP server. Servers can retrieve cookies from the client to access state information about exchanges with the client in a current or previous session.

Cookies were originally designed and specified by Netscape. This specification is not part of the HTTP protocol, but is managed by Netscape. It is available at [http://www.netscape.com/newsref/std/cookie\\_spec.html](http://www.netscape.com/newsref/std/cookie_spec.html).

When a server wants to preserve session state information it includes in the response a Set-Cookie header with the cookies and values it wants to preserve. For example:

```
Set-Cookie: Customer=Cosmo; Path=/raggs; Expires: Tue, 04 Dec 2002 17:31:14 GMT
```

An iPlanet UDS HTTP server application uses the Cookie class to create cookies and add them to a response. It first invokes the Cookie.SetName method to create the cookie "Customer":

```
SetName('Customer');
```

The server then invokes the Cookie.SetValue method to set the Customer cookie value to "Cosmo":

```
SetValue('Cosmo');
```

The server repeats this sequence for any additional cookies it needs to process. Finally, the server invokes the HTTPBaseResponse.AddCookie method to add the cookies to the request:

```
cookie: cookie = new;
cookie.SetName('Customer');
cookie.SetValue('Cosmo');
cookie.SetPath('/raggs');
cookie.SetExpiration('04 Dec 2000 14:21:03 GMT');
AddCookie('cookie')
```

---

**NOTE** The HTTPSupport library automatically manages headers for client and server applications that send cookies.

---

## Message Headers

Just as with client requests, a server response includes message headers that provide the client with information about the message. Typically, the HTTPSupport library automatically creates message headers for you. But you can use the following methods, available from HTTPBaseRequest, to create or add headers.

```
SetHeader(string name, string value)
SetIntHeader(string name, integer value)
AddHeaderValue(string name, string value)
```

SetIntHeader creates headers that take integer value, (for example, SetIntHeader('max-forwards', 5). Add HeaderValue creates multiple headers with the same name.

## Message Body

The response from an HTTP server includes a message body, which could be empty. The section [“Message Body” on page 555](#), which is about client requests, also applies to server responses. Use the HTTPBaseMessage.SetBody method to replace the default entity (an empty body) with one of your own. The syntax of SetBody is:

```
SetBody(GenericEntity body)
```

*body* is the entity you want to specify.

The body of a message is an object that implements the GenericEntity interface. The GenericEntity interface provides the following:

- methods to read from and write to a memory stream
- get and set methods to access the headers of the body
- methods to compose entities

[Code Example 18-5](#) provides some examples of how you can build the body of a message.

### Code Example 18-5 Processing the body of a message

```
DoPost(request : HTTPBaseRequest, response : HTTPBaseResponse)
Begin
    Txt : TextData = new;
    Request.body.readLine(txt);
    . . .
End;
```

**Code Example 18-5** Processing the body of a message (*Continued*)

```

DoGet(request : HTTPBaseRequest, response : HTTPBaseResponse)
Begin
    . . .
    Response.body.WriteLine('<H1> Big smiley :) </H1>');
End;

```

## Entities

The Entity class provides the default entity for containing data in HTTP messages. It provides methods for writing various kinds of data from the memory stream into an entity, setting the content type of the entity, and nesting entities. You can extend and customize the Entity class to accommodate different MIME types.

### *Multi-part Messages*

When a multi-part message is created, the data of each type is put in a separate entity, all of which are contained in an “entity holder” entity. For example, if you created a message whose body contained HTML, XML, and JPEG data, the body contains four entities, one for each data type plus the entity holder. Each entity has its own Content-Type and Content-Length headers.

To add entities to the entity holder, use the Entity.AddEntity method. For example:

```
AddEntity('text/xml');
```

adds an entity containing XML data to the current entity. You need to call AddEntity for each entity you want to add.

---

**NOTE** An entity can be empty, contain data, or contain other entities. You cannot add an entity to the current entity if the current entity already contains data.

---

### *Nesting Entities Within Entities*

Messages that include quoted copies of previous messages (including forwards of messages to new recipients) are represented as entities nested within other entities. The current message is the outer entity and the quoted or forwarded message is the inner entity. Each successive generation of quoting or forwarding adds a new inner entity to the innermost current entity.

To nest entities, invoke the AddEntity method on each successive entity (rather than repeatedly invoking AddEntity on the same current entity).

## HTTPFactory

The HTTPFactory object is responsible for the instantiation of HTTP entities and messages. Using an HTTPFactory object, you can register your own entity classes (which must implement the GenericEntity interface) or HTTP messages (which must extend either HTTPBaseRequest or HTTPBaseResponse).

### *Registering Entity Subclasses*

By default, an entity object in HTTPSupport is text/html. When a message is received, whatever its mime type, the entity object is used with its content type set to the message body content type. [Code Example 18-6](#) shows how to register a user-defined entity (XMLEntity) using the HTTPFactory.

#### **Code Example 18-6** Registering a user-defined entity using HTTPFactory

```
helper : HTTPHelper = new;
factory : HTTPFactory = helper.findFactory();

factory.addMimeType('text/xml', XMLEntity);
```

### *Registering HTTP Messages*

Similarly, you can register your own class of HTTP request or response (which extends either HTTPBaseRequest or HTTPBaseResponse). [Code Example 18-7](#) shows how to register UserRequest, which extends HTTPBaseRequest.

#### **Code Example 18-7** Registering a user-defined HTTP request

```
helper : HTTPHelper = new;
factory : HTTPFactory = helper.findFactory();

factory.registerClass(UserRequest);
```

## Using Multi-Threaded Server Processes

You can create multiple server instances to have a pool of servers available to run HTTP sessions concurrently. When a client sends a request, the request is routed to the first available server instance. If all instances are busy, the request is queued for dispatch to the first available server.

---

**NOTE** When a client opens a persistent network session with a server, all requests from that client during that session are dispatched to the same server instance. For non-persistent sessions, a client is not guaranteed connection to the same server instance.

---

Use the `HTTPHelper.Advertise` method to configure the maximum number of server instances:

```
Advertise('instances=n')
```

*n* is the maximum number of server instances you want to create. The default value for *n* is one.

There is no upper limit on the number of server instances or client connections you can configure. However, you are constrained by the amount of memory allocated at the operating system level for the iPlanet UDS process.

## Configuring HTTP Sessions

Sessions created in iPlanet UDS server and client applications contain default configurations. However, you can override the default configuration using methods provided by the `HTTPConfigManager`, `HTTPSession`, and `HTTPHelper` classes. The process for configuring servers and clients differ, as illustrated in the following sections.

### Configuring HTTP Clients

Sessions created for iPlanet UDS HTTP clients are automatically configured with default values. However, you can use the `HTTPConfigManager` to override the default values with your own. Subsequent sessions contain the new values. To specify configuration on a session-by-session basis, use the `HTTPSession` class.

You typically configure iPlanet UDS HTTP client applications to do the following:

- Specify the behavior for the client and server when establishing an HTTP application session
- Specify persistence for network sessions
- Specify that the session is a secure HTTP session.

When configuring secure sessions, you can then set levels for secrecy, integrity, and authentication as well as configuration parameters for certificates and other SSL features

### ► To configure an iPlanet UDS HTTP client

1. Before creating a request object, use `HTTPConfigManager.SetConfigValue` to override default application, network, or SSL configuration options.

The values you specify become the new default values for subsequent HTTP sessions.

2. Build the request object, which contains an HTTP session instance.
3. Use `HTTPSession.SetSessionValue` to override configuration options, specifying values valid for the current session only
4. Dispatch the request

After dispatching the request, the HTTP client negotiates the session with the HTTP server. [Code Example 18-8](#) provides an example for configuring iPlanet UDS HTTP client applications.

#### Code Example 18-8 Configuring an iPlanet UDS client application

```
-- Before creating the server object,
-- set the configuration to create secure sessions
configMgr : HTTPConfigManager = HTTPHelper().FindConfigManager();
configMgr.SetConfigValue(SSL_SECURE, SSL_ALWAYS);

-- Now build the client request
request : HTTPBaseRequest = new;
request.setMethod('GET');
request.setURL('http://iplanet.com:8443');

-- Before sending, specify specify session-specific values
session : HTTPSession = new;
request.SetSession(session);
session.setSessionValue(SSL_AUTHENTICATE, SSL_ANONYMOUS);
```



## System-Wide Configuration

To override the default configuration use the `SetConfigValue` method from the `HTTPConfigManager` class:

```
SetConfigValue(integer config, integer value)
```

*config* specifies a configuration option

*value* specifies the value for the option.

Use `SetConfigValue` before your application creates a request object. [Table 18-4](#) lists the options you can configure, including their default values, for `SetConfigValue`.

**Table 18-4** Configuration options for iPlanet UDS HTTP client applications

config Option	Possible values	Description
HTTP_CONFIG_TCPSESSION_VALUE	HTTP_TCPSESSION_PERSISTENT HTTP_TCPSESSION_MESSAGE	Specifies persistence for network sessions. <ul style="list-style-type: none"> <li>• <i>Persistent</i> sessions remain open until either the client or server explicitly close the session. (Default)</li> <li>• <i>Message</i> sessions are opened and closed for each message sent or received.</li> </ul>
HTTP_CONFIG_SESSION_VALUE	HTTP_SESSION_MAYBE HTTP_SESSION_NONE HTTP_SESSION_REQUIRED	Specifies how client and servers establish application sessions. <ul style="list-style-type: none"> <li>• <i>Maybe</i> specifies that the client lets the server decide if a session needs to be established. (Default)</li> <li>• <i>None</i> specifies that no application session can be established.</li> <li>• <i>Required</i> specifies that the application session is mandatory.</li> </ul>

**Table 18-4** Configuration options for iPlanet UDS HTTP client applications (*Continued*)

config Option	Possible values	Description
SSL_SECURED	SSL_NEVER SSL_ALWAYS	<i>Never</i> establishes a non-secure session. (Default)  <i>Always</i> establishes a secure session. For information on configuring secure sessions, refer to “ <a href="#">Configuring Secure Sessions</a> ” on page 575.

## Session-By-Session Configuration

To override the configuration settings for clients on a session-by-session basis, use the `SetSessionValue` method from the `HTTPSession` class:

```
SetSessionValue(integer config, integer value)
```

*config* specifies a configuration option

*value* specifies the value for the option.

Use `SetSessionValue` after your application creates a request object, but before dispatching a request. The options for `SetSessionValue` are the same as those listed for `SetConfigValue` listed in [Table 18-4 on page 569](#).

## Configuring HTTP Servers

Sessions created for iPlanet UDS HTTP servers are automatically configured with default values. However, you can use the `HTTPConfigManager` to override the default values with your own. You also specify configuration for iPlanet UDS HTTP servers using the `HTTPHelper` class and the `HTTPServerManager` to configure request dispatching.

You typically configure iPlanet UDS HTTP server applications to do the following:

- Specify the behavior for the client and server when establishing an HTTP application session
- Specify persistence for network sessions
- Set the port, the number of server instances, and the timeout period
- Specify that a session is a secure HTTP session

When configuring secure sessions, you can then set levels for secrecy, integrity, and authentication as well as configuration parameters for certificates and other SSL features. For information on configuring secure sessions, refer to [“Configuring Secure Sessions” on page 575](#).

► **To configure an iPlanet UDS HTTP server**

1. Make sure the application implements either the `MessageReceiver` or `HTTPReceiver` interface.
2. Use `HTTPConfigManager.SetConfigValue` to override default configuration options.

Call `SetConfigValue` after you have created the server object.

3. Use `HTTPHelper.Advertise` to set up the HTTP server to listen for HTTP messages.

You can override default values for the port, the number of objects to dispatch requests, and the timeout period. If you specified a secure server with a call to `SetConfigValue` can also specify secure options in the call to `Advertise`.

4. Access the session object in the `Initialize` method to specify session-specific behavior.

Use the pre-allocated `MessageConfig` object to access the session in the `Initialize` method. For secure HTTP servers, you can set (or override) secure configuration options setup either by default values or by previously using `SetConfigValue`

After calling `Advertise`, the HTTP server listens for HTTP messages and can establish sessions with HTTP clients. [Code Example 18-9](#) provides an example for configuring an iPlanet UDS server application:

**Code Example 18-9** Configuring an iPlanet UDS server application

```
-- In method of server appl. that creates the server object
. . .

-- Instantiate a server object
server : myServer = new;

-- Configure a persistent session
configMgr : HTTPConfigManager = HTTPHelper().FindConfigManager();
configMgr.SetConfigValue(HTTP_CONFIG_TCPSESSION_VALUE,
                        HTTP_TCPSESSION_PERSISTENT);

-- Override the default value for a secure port
HTTPHelper().Advertise( server, 'port=8080' );
```

**Code Example 18-9** Configuring an iPlanet UDS server application (*Continued*)

```

. . .
-----
-- Now in the Initialize Method, which
-- takes a pre-allocated config object
method MyHttpServer.Initialize(input config:
                                HTTPSupport.MessageConfig)
begin
. . .

-- Get the session object
session : GenericSession = config.GetSession();

-- Specify the type of session
session.SetNetworkSession( HTTP_TCPSESSION_MESSAGE );
. . .
end method

```

## HTTPConfigManager

For configuring iPlanet UDS HTTP servers, the SetConfigValue method from the HTTPConfigManager class has the following syntax:

```
SetConfigValue(integer config, integer value)
```

*config* specifies a configuration option

*value* specifies the value for the option.

Use SetConfigValue before your server application calls HTTPHelper.Advertise. The options for configuring iPlanet UDS HTTP servers are the same as those listed for configuring iPlanet UDS HTTP clients listed in [Table 18-4 on page 569](#).

## HTTPHelper

To enable your HTTP server application to listen for messages, call HTTPHelper.Advertise within an event loop.

Advertise has the following syntax:

```
Advertise(object obj, string param)
```

*obj* is the server object you are configuring

*param* is a string containing one or more *name=value* pairs.

For example:

```
Advertise(my_server, 'port=80 instances=12 timeout=120')
```

configures the server *my\_server* to listen for requests on port 80, instantiates 12 instances of *my\_server*, and sets the maximum idle time for *my\_server* to two minutes.

**Table 18-5** lists the options for `Advertise` that you can configure as name/value pairs, and also lists their default values.

**Table 18-5** Configuration options for `Advertise`

Parameter	Description
Port	Sets the port on which the server listens for requests. (Default <i>port=80</i> for non-secure servers, <i>port=443</i> for secure servers)
Instances	Sets the number of server instances used to dispatch requests. (Default is <i>instances=1</i> .)
Timeout	Sets the maximum idle time the server holds the connection of an application session. (Default is <i>timeout=0</i> , indicating no timeout.)

## Configuring Request Dispatching With `HTTPServerManager`

`HTTPSupport` uses its dispatcher to route requests to available server instances. The dispatcher maintains a list of server instances that are currently associated with client requests, and another list of server instances that are currently free. Once a request is routed to a server instance, that instance is moved from the “free” list to the “booked” list until either the request ends or the application session between the client and that server instance is terminated. At that point the instance is moved back to the “free” list.

There is one dispatcher per `Advertise` method call. To find a particular dispatcher instance, you need the object used in the `Advertise` method call. The `HTTPHelper.FindServerManager` method returns the server manager associated with a particular advertised object. For example:

```
helper : HTTPHelper = new

//The 'obj' parameter is the object used in the Advertise call
mgr : HTTPServerManager = helper.FindServerManager(obj);

busy : integer = mgr.activeServers();
available : integer = mgr.freeServers();
...
```

The HTTPServerManager interface provides the following methods:

Method	Description
ActiveServers	The ActiveServers method returns the number of HTTP server object instances in use (either booked by an application session or used during a simple HTTP request/response).
AddInstances	<p>The AddInstances method creates new HTTP server object instances. The new object is immediately added to the pool of free HTTP server objects. The original object used for the HTTPHelper.Advertise method call is used to instantiate the new objects. Syntax for AddInstances is:</p> <p>AddInstances(<i>n</i>)</p> <p>where <i>n</i> is the number of new HTTP server instances to create.</p>
CloseApplicationSession	<p>The CloseApplicationSession method explicitly releases a session. The object associated with this session returns immediately into the pool of the available free HTTP server objects. The syntax for CloseApplicationSession is:</p> <p>CloseApplicationSession(<i>id</i>)</p> <p>where <i>id</i> is the session ID value to close.</p>
FreeServers	The FreeServers method returns the number of HTTP server objects available. The sum of the ActiveServers and FreeServers values is the total of HTTP server object instances.
GetApplicationSessions	The GetApplicationSessions method returns the list of all the sessions that are currently in use. A session can be either an application session explicitly established between the client and the server, or a temporary session used to book the server object at the time of the client request.
GetInstances	The GetInstances method call returns the list of all the HTTP server object instances. The number of objects returned by this method call is the sum of the values returned by the ActiveServers and FreeServers methods.

Method	Description
RemoveInstances	<p>The RemoveInstances method removes object instances from the pool of free HTTP server objects. The value of the number parameter specifies how many instances to remove.</p> <p>If the number specified is higher than the number of objects in the free server object pool, the remainder of objects are freed later under one of the following circumstances:</p> <ul style="list-style-type: none"> <li>• when objects become free again</li> <li>• at the end of a session</li> <li>• at the end of a request/response cycle</li> </ul> <p>The syntax for RemoveInstances is:</p> <p>RemoveInstances(<i>n</i>)</p> <p>where <i>n</i> is the number of HTTP server instances to remove.</p>
SetTimeout	<p>The SetTimeout method sets the timeout value associated with the application sessions. The timeout is expressed in seconds. If an application session is not used for the duration of the timeout, the session is released. The syntax for SetTimeOut is:</p> <p>SetTimeOut(<i>seconds</i>)</p> <p>where <i>seconds</i> is the length in seconds of the timeout.</p>

## Configuring Secure Sessions

iPlanet UDS HTTP server and client applications can be configured to implement secure sessions that use the Secure Sockets Layer (SSL) protocol. For information on secure sessions and the SSL protocol, refer to [“Secure Sessions” on page 550, Chapter 19, “Enabling Security,”](#) and the iPlanet UDS online help.

### Secure Client Sessions

To configure secure HTTP clients, use the same process described in [“Configuring HTTP Clients” on page 567](#). For system-wide configuration, use `HTTPConfigManager.SetconfigValue`; for session-by-session configuration use `HTTPSession.SetSessionValue`.

► **To configure a secure client session**

1. Specify that all sessions created in your application are secure by calling `SetConfigValue` in your application *before* creating a request object:

```
configMgr : HTTPConfigManager =
    HTTPHelper().FindConfigManager();
configMgr.SetConfigValue(SSL_SECURE, SSL_ALWAYS);
```

or

1. Specify that an individual session is secure by calling `SetSessionValue` in your application *after* creating a request object:

```
session : HTTPSession = new;
request.SetSession(session);
session.SetSessionValue(SSL_SECURE, SSL_ALWAYS);
```

After enabling a secure session, the SSL options listed in [Table 18-6](#) are enabled. You can then call `SetConfigValue` and `SetSessionValue` to override any default settings.

**Table 18-6** SSL options for `SetConfigValue` and `SetSessionValue`

config Option	Possible values	Description
SSL_SECRECY	SSL_ANY SSL_NONE SSL_LOW	<p>SSL_MEDIUM SSL_HIGH</p> <p>Specifies the level of secrecy.</p> <p>If an endpoint specifies <i>any</i>, then the other endpoint sets the value. (Default)</p> <p>If both endpoints specify <i>any</i>, then <i>high</i> is the level of secrecy.</p>
SSL_INTEGRITY	SSL_ANY SSL_NONE SSL_LOW	<p>SSL_MEDIUM SSL_HIGH</p> <p>Specifies the level of integrity.</p> <ul style="list-style-type: none"> <li>• If an endpoint specifies <i>any</i>, then the other endpoint sets the value. (Default)</li> <li>• If both endpoints specify <i>any</i>, then <i>high</i> is the level of integrity.</li> </ul>
SSL_RESUME	SSL_NEVER SSL_ALWAYS	<p>Specifies whether to automatically resume connections.</p> <ul style="list-style-type: none"> <li>• <i>Always</i> specifies always resume the connection. (Default)</li> <li>• <i>Never</i> specifies never resume the connection.</li> </ul>



**Table 18-6** SSL options for SetConfigValue and SetSessionValue (Continued)

config Option	Possible values	Description
SSL_AUTHENTICATE	SSL_SERVER SSL_ANONYMOUS SSL_NONE SSL_BOTH	<p>Specifies how a session is authenticated.</p> <ul style="list-style-type: none"> <li>• <i>Server</i> specifies authentication for server only. (Default)</li> <li>• <i>Anonymous</i> specifies to use a negotiated key.</li> <li>• <i>None</i> specifies no authentication.</li> <li>• <i>Both</i> specifies authentication for both server and client.</li> </ul>

A session that has secure sessions enabled can also specify certificate settings using the following syntax for SetConfigValue and SetSessionValue:

```
SetConfigValue(integer config, string value)
```

```
SetSessionValue(integer config, string value)
```

*config* specifies a configuration option

*value* is string specifying a pathname or password.

**Table 18-7** lists the options for configuring certificates.

**Table 18-7** Certificate configuration settings for secure sessions

Configuration Option	Description
SSL_CERTFILE	String specifying the path to a certificate file
SSL_CERTPASSWORD	String specifying the certificate password
SSL_CERTPASSWORDFILE	String specifying the path to the certificate password file
SSL_ROOTFILE	String specifying the path to the root of certificates file
SSL_ROOTPASSWORD	String specifying the root of certificates password
SSL_ROOTPASSWORDFILE	String specifying the path to the root of certificates password file

## Secure Server Sessions

To configure secure HTTP servers, use the same process described in “[Configuring HTTP Servers](#)” on page 570.

### ► To configure a secure HTTP session

1. In an HTTP server application, first enable secure sessions by enabling security using the configuration manager:

```
configMgr : HTTPConfigManager =
                HTTPHelper().FindConfigManager();
configMgr.SetConfigValue(SSL_SECURE, SSL_ALWAYS);
```

This enables the configuration options listed in [Table 18-6 on page 576](#). You can then use `SetConfigValue` to override the default secure settings.

2. Specify certificate settings using the version of `SetConfigValue` with the following syntax:

```
SetConfigValue(integer config, string value)
```

*config* specifies a configuration option and *value* is string specifying a pathname or password. [Table 18-7 on page 577](#) lists the options for configuring certificates.

3. Finally, use `HTTPHelper.Advertise` to override the default port setting or to override other secure configuration options.

You use *name=value* pairs to specify the secure configuration options to `Advertise`. For example:

```
Advertise(my_server,
          'Instances=12 Timeout=240 Integrity=HIGH');
```

[Table 18-8](#) lists the secure configuration options available for `Advertise`, including their default values. Refer to [Table 18-6 on page 576](#) for a description of these secure settings.

**Table 18-8** Secure Session Configuration Options for `Advertise`

Parameter	Description
Secure	SSL_Always, SSL_Never (Default, <i>Secure=SSL_Never</i> )
Secrecy	None, Low, Medium, High, Any (Default, <i>Secure=Any</i> )
Integrity	None, Low, Medium, High, Any (Default, <i>Secure=Any</i> )
Compression	None, Speed, Size, Any (Default, <i>Compression=None</i> )
Resume	Never, Always (Default, <i>Resume=Always</i> )
Authenticate	Anonymous, Server, Both, Any (Default, <i>Authenticate=Anonymous</i> )

## Related Topics

Depending on the nature of your application you may find the following topics useful.

### Encoding and Decoding With Base64

To ensure safe transport of various data types across 7-bit networks you can encode the data using the `HTTPBaseMessage.EncodeBase64` method:

```
encodeBase64 ( 'plainText' )
```

*plainText* is the data you want encoded. The output of this method is a string of encoded data.

To decode Base64-encoded data, use the `HTTPBaseMessage.DecodeBase64` method:

```
DecodeBase64 ( 'encodedText' )
```

*encodedText* is the data you want to decode. The output of this method is a string of decoded data.

### Character Sets in Messages

By default, iPlanet UDS uses whatever character set you have configured at the operating system level. For instance, if your operating system is configured for Latin-1, that is what iPlanet UDS uses.

iPlanet UDS has the ability to do some degree of character set conversion on its own. Generally, this means that if the character sets map cleanly to each other, then iPlanet UDS can do the conversion properly. However, in cases where one character set has characters for which there is no corresponding character in the other set, iPlanet UDS converts what it can and those characters that do not map cleanly may be represented as meaningless data. This is most likely to happen when converting multi-byte character sets to 7-bit sets.

If your application spans locales that use different character sets, and one of those character sets is a superset of all of them, you should use that one in all locales if practical. Doing this means that no conversion needs to be done at all. However, there may be cases where none of the character sets being used is a complete superset of them all. In such cases, your best bet is to use the most inclusive set and be aware that some data may not be able to be represented in the less inclusive character sets.

## Related Topics

# Enabling Security

This chapter introduces the classes that implement Secure Sockets Layer (SSL) services. SSL is a standard security protocol that uses encryption and authentication techniques to protect communication on corporate intranets and internets. iPlanet UDS's runtime services support the SSL connection as well as the building of certificates for encryption and authentication.

A detailed description of each SSL class is provided in online help.

## About SSL

SSL (Secure Sockets Layer) is a security protocol that protects communication on corporate intranets and internets. Originally developed by Netscape, SSL is a de facto standard supported by most Web browsers and servers.

Constructed as a separate layer within the Internet protocol architecture, SSL itself consists of two main layers, a data layer and a record layer. The data layer is where SSL messages are created and the record layer formats and frames the messages, passing them to the underlying transport layer for transmission.

The Transmission Control Protocol (TCP) provides secure transport and protects data transferred over various Internet protocols, including Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Net New Transfer Protocol (NNTP).

SSL secures data in three important ways by providing:

- Secrecy of the connection. An initial handshake defines a secret key. Symmetric cryptography, in which both parties must construct and share a secret key, is then used for data encryption (for example, DES or RC4).

- Authentication of senders and receivers. Asymmetric or public key cryptography (with a public/private key pair) is used for authentication (for example, using the RSA standard).
- Message integrity (accurate and complete preservation of message content). Message digests (MD5 or SHA) are used for MAC (Message Authentication Code) computations and to detect tampering.

An extensible framework in SSL enables new public key and encryption methods to be incorporated as needed.

## How SSL Works

SSL support in Web browsers is mainly transparent to the end user. Netscape and Microsoft Internet Explorer browsers display lock icons in the status bar at the bottom of the window. The icon appears locked when the client is using an SSL-secured web site and unlocked when the connection is not secured. Web sites that require SSL display HTTPS rather than HTTP in the prefix of the URL, because SSL is the underlying mechanism for HTTPS. Browsers are typically preloaded with public keys from certificate authorities. The client system uses one pair of keys to secure the data it sends.

On the server, SSL must be enabled and the server supplied with public key certificates from the same certificate authorities used by the client systems. You set up the server to use these certificates. The server uses its own pair of keys to secure the data it sends.

The client and server prepare for SSL-secured communications by stepping through the SSL handshake protocol, which allows the client and server to authenticate each other and negotiate an encryption algorithm and keys before the application sends or receives any data. The handshake protocol coordinates client and server states as the messages are exchanged and processed.

When the protocol completes successfully, data can be transmitted. SSL fragments the data into manageable blocks, applies a MAC, encrypts the data, and sends the result. At the other end, the data is decrypted, verified, reassembled, and then delivered to end clients as appropriate.

# SSL Services

An iPlanet UDS application can secure communications by using SSL with an external process or program that is running locally or on another host. An iPlanet UDS application can exchange data with a Web browser, a Java, C, or BASIC program, telnet, HTTP, and so on.

UDS complies with and defaults to SSL 3.0. For details of the SSL 3.0 specification, see <http://home.netscape.com/eng/ssl3/draft302.txt>.

UDS supports the following cryptography standards, which are used in many SSL implementations:

- X.509 certificates in ASN1.0/DER (Abstract Syntax Notation Digital Encoding Rules)
- PKCS #7 format
- PKCS #12 format
- The signature algorithms RSA (Rivest Shamir Adelman) and DSA (Digital Signature Algorithm)
- The message digests MD5 (Message Digest 5) and SHA-1 (Secure Hash Algorithm)

TOOL supports the following types of certificates:

- Client SSL certificates to identify clients to servers. Client authentication is optional in SSL sessions.
- Server SSL certificates to identify servers to clients. Server authentication is required for an SSL session.
- Certificate authority certificates, also known as AcceptableRoot certificates. This is a set of root or CA certificates that are considered trustworthy. If the certification path for a certificate leads to a certificate in this list, the connection is allowed.

SSL support is provided for HTTP client or server applications that use the iPlanet UDS HTTPSupport library. For more information, refer to [Chapter 18, “Creating HTTP Applications” on page 543](#). The UDS online help also contains information on building secure HTTP applications.

# SSL Classes

SSL services available from iPlanet UDS provide security for the connection by:

- Returning the attribute values negotiated between a client and server during the initial SSL handshake. These values indicate the SSL version, whether the open connection can be resumed, the secrecy and integrity algorithms, the certificates and certificate roots, and the connection states.
- Letting you set automatic renegotiation of security parameters at a fixed time interval

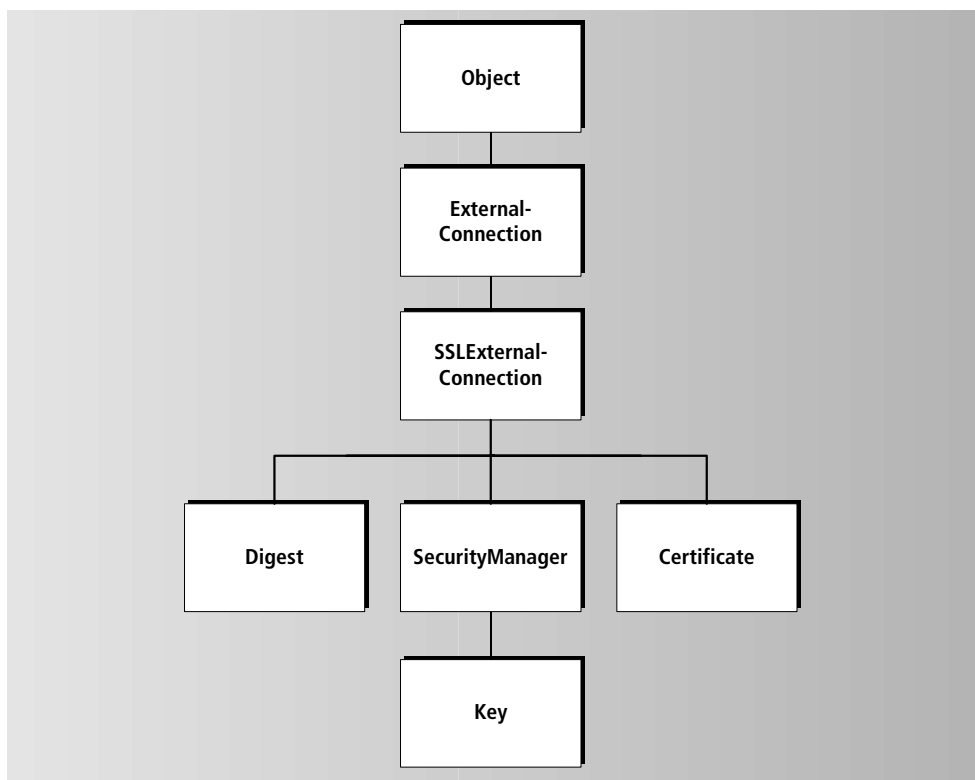
Services related to certificates include:

- Reading and writing of certificate contents
- Creating and verifying message digests
- Identifying the public/private key pair type (for example, RSA for key exchange or RSA for digital signing)
- Basic management and testing of certificates

SSL is supported through enhancements to ExternalConnection. A UDS application uses the ExternalConnection class to provide communications with an external process or program that is running locally or on another host.

Classes that support SSL and the inheritance hierarchy that determines their relation are shown in [Figure 19-1](#):



**Figure 19-1** SSL Class Hierarchy

The following table lists and describes the classes that provide SSL services. Each SSL class is explained in detail in the online help.

<b>Class</b>	<b>Description</b>
SSLEternalConnection	Adds security for network communications between two points (peer to peer or client to server) when one of the points is a UDS application. Attribute values indicate the SSL version, whether the open connection can be resumed, the secrecy and integrity algorithms, the certificates and certificate roots, and the connection states.
SecurityManager	Provides runtime security services and basic certificate testing and management. You can use these services in conjunction with your own certificate management system.

Class	Description
Certificate	Enables the creation and reading of X.509 certificate contents. The contents include SSL version and various certificate properties: serial number, algorithm identifiers, issuer, validity period, subject, subject's public key, optional subject and issuer unique IDs and optional extensions.
Key	Contains the state information for a public/private key pair and identifies them as one of the following: RSA_KEYX (for key exchange only), RSA_SIGN (authentication with RSA digital signature), or DSA_SIGN (authentication with DSA digital signature).
Digest	Provides the state information needed for creating and verifying the message digest or fingerprint.

## Working with Certificates

The SecurityManager class provides methods for importing, exporting, issuing, and verifying certificates. It contains additional methods for working with message digest objects and generating keys. For more information on the SecurityManager class, refer to online help.

iPlanet UDS provides the ossecdrv diagnostic utility, which you can use for debugging or for creating certificates for testing purposes. The ossecdrv utility is in the following location:

```
$FORTE_ROOT/install/diag/bin
```

---

**CAUTION** The ossecdrv utility is provided for testing and debugging purposes only. It is not designed for use in a production environment.

---

## Creating a Root Certificate

The following script creates a root certificate, which can be used for testing purposes. Save this script as a file (for example, CreateRootCert.csc) and use it as input to ossecdrv.

```
ossecdrv -i CreateRootCert.csc
```

### Code Example 19-1 Creating a root certificate for testing SSL

```
#
# Generate a root certificate for test purposes
# Save the cert and key in rootkey.dat
# Save the cert in root.dat
# Save the root in roots.dat
#
GenerateKey RSA_SIGN
CreateCertificate
SetCertNotBefore 01-jan-1998
SetCertNotAfter 01-jan-2009
SetCertSerialNumber 0x01
SetCertSubject country US
SetCertSubject organization "Sun Microsystems Inc."
SetCertSubject organizationunit1 "Forte Tools"

SetCertKeyUsage 6
SetCertBasicConstraints 0
SetCertKey current
IssueCertificate
VerifyCertificate
ExportCertificate root
ExportWrite rootkey.dat
ExportCertificate
ExportWrite root.dat
AddCertificateToSet
SaveCertificate
ExportSetOfCertificates roots
ExportWrite roots.dat
```

## Creating a Leaf Certificate

The following script creates a leaf certificate, which can be used for testing purposes. A leaf certificate is any certificate in the certificate hierarchy other than the root certificate.

Save this script as a file (for example, CreateLeafCert.csc) and use it as input to ossecdrv.

```
ossecdrv -i CreateLeafCert.csc
```

### Code Example 19-2 Creating a leaf certificate for testing SSL

```
#
# Generate a leaf certificate for test purposes
# Save the cert and key in leafkey.dat
# Save the cert in leaf.dat
#
GenerateKey RSA_SIGN
CreateCertificate
SetCertNotBefore 01-jan-1999
SetCertNotAfter 01-jan-2008
SetCertSerialNumber 0x01
SetCertSubject country US
SetCertSubject organization "Sun Microsystems Inc."
SetCertSubject organizationunit1 "Forte Tools"
SetCertSubject organizationunit2 "QA"
SetCertSubject name "Leaf Certificate"
SetCertSubject locality "Oakland"

SetCertKey current
IssueCertificate saved
VerifyCertificate
ExportCertificate leaf
ExportWrite leafkey.dat
ExportCertificate
ExportWrite leaf.dat
```

# Code Examples

This section contains code examples that show how a client can communicate with a secure server.

**Code Example 19-3** shows how a client can request server authentication and read the certificate contents.

## **Code Example 19-3** Requesting server authentication

```

ec : SSLEExternalConnection = new;
ms : MemoryStream = new;
certificate : Certificate;
rLen : i4;
wLen : i4;

-- We want the server to be authenticated.
ec.Authenticate = SSLEExternalConnection.SERVER;

-- Try something. Equivalent to https://www.sun.com/
ec.Open('www.sun.com', 443, Nil, 0);

-- Take a look at the servers certificate chain.
certificate = ec.OtherCertificate;

-- Request the default.
ms.Open(SP_AM_READ_WRITE, FALSE);
ms.WriteText('HTTP_1.0\nGET \n\n');
wLen = ms.Size;
ec.Write(ms, wLen);

-- Read back the response
while TRUE do
    rLen = 8192;
    ec.Read(ms, rLen);
    if (rLen = 0) then
        exit;
    end if;

    -- Process rLen characters of the response here.
end while;

```

**Code Example 19-4** shows how a secure server can supply its certificate for authentication and read anything sent back by the client. First, an authenticated session is established with the server supplying its certificate. The server then listens for connections, reading requests and writing responses for each connection.

**Code Example 19-4** Secure server supplying certificate for authentication

```

ec : SSLExternalConnection = new;
c  : ExternalConnection;
ms : MemoryStream = new;
cert : Certificate;
rLen : i4;
wLen : i4;

-- We want SERVER authenticated connections
ec.Authenticate = SSLExternalConnection.SERVER;

-- Here is the servers certificates (setup elsewhere)
ec.MyCertificate = cert;

-- For each connection
while TRUE do
    c = ec.StartListening(443);

    -- Read the request.
    ms.Open(SP_AM_READ_WRITE, FALSE);
    rLen = 8192;
    c.Read(ms, rLen);
    ms.Close;

    -- Write a response.
    ms.Open(SP_AM_READ_WRITE, FALSE);
    ms.WriteText('401');
    wLen = ms.Size;
    c.Write(ms, wLen);
    ms.Close;
    c.Close;
end;

```

**Code Example 19-5** shows how to set up an array of certificates as an `AcceptableRoots` certificate.

**Code Example 19-5** Setting up an `AcceptableRoots` certificate

```

ec : SSLEExternalConnection;
c  : Certificate;
a  : Array of Certificate;
b  : BinaryData;
s  : TextData;

-- Create an array of certificates. Used as the AcceptableRoots
-- of a Certificate.

-- Read a certificate from a file into a BinaryData.
-- The certificate file can be created by exporting
-- a root certificate from a browser.

-- Normally a certificate does not need to be encrypted if
-- it does not include a private key.
c = task.Part.SecMgr.ImportCertificate(b, Nil);

-- Add a certificate to an array.
a.AppendRow(c);

-- When you have all the certificates in the array then
-- export the set of certificates. This is usually exported
-- with a password so that it cannot be tampered with.
-- Write the result to a file for future use.
b = task.Part.SecMgr.ExportCertificates(a, s);

-- The reverse is a little easier. Read the certificates file
-- into a BinaryData including the password.
a = task.Part.SecMgr.ImportCertificates(b, s);

-- Associate the set of certificates
-- with an SSLEExternalConnection
ec.AcceptableRoots = a;

-- AcceptableRoots are checked by a client to determine if
-- certificates being used by the server should be trusted
-- by us. AcceptableRoots are checked by a server in a similar
-- manner when a client is being authenticated.

```





# Using the XMLDOM2 Library

The Document Object Model (DOM) is an API that provides a programmatic way of manipulating objects within well-formed XML documents. Using DOM, programmers can navigate the logical structure of a document, and can add, modify, and delete document objects and content. The iPlanet UDS XMLDOM2 class library supports the core functionality of the DOM Level 2 specification, as described in <http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/>.

This document covers the following topics:

- XML namespaces, on [page 594](#)
- Tree-based APIs, on [page 595](#)
- The Distributed Object Model, on [page 595](#)
- The XMLDOM2 classes, on [page 600](#)
- DOM Level 2 features, on [page 601](#)
- Upgrading from DOM Level 1, on [page 602](#)

## What Are XML Namespaces?

XML namespaces are used to qualify attribute and element names in XML documents, by associating them with URI references that identify unique namespaces. This allows applications to distinguish similarly named elements and attributes from different XML vocabularies, even if they are mixed together in the same document. These names are shortened by using prefixes. Prefixes are bound to namespaces by using the `xmlns` attribute.

A prefixed name is referred to as a *qualified name*. The two components of a qualified name are the prefix and the local name. For example:

```
<xsl:output xmlns:xsl="http://www.w3.org/1999/XSL/Transform" />
```

In this example, the first part “xsl” is the prefix, and the second part “output” is the local name. Together they form the qualified name.

When DOM elements and attributes are created they are permanently associated with a namespace URI.

Neither the validity of a namespace, nor the mapping between namespace prefixes and URIs, is checked by the DOM. These functions must be implemented at the application level. The DOM also does not perform any validation on namespace prefixes and URIs; it assumes both to be properly formed, and it is the responsibility of the application to ensure that they are.

# Tree-Based APIs

XML parsers operate in one of two ways: either they generate a series of events, each corresponding to the parsing of a single XML syntactic unit (start tag, datum, end tag, etc.), or they create a tree representing the hierarchy of elements in the XML document. The DOM does the latter, which is called tree-based parsing.

## Advantages of Using Tree-Based APIs

The primary advantage of tree-based parsing is that the resulting tree represents the entire document, and once the tree has been constructed every parse node on the tree is immediately available. For applications where specific nodes are manipulated tree-based parsing is very efficient.

## Restrictions When Using Tree-Based APIs

Tree-based parsing has two major drawbacks: First, for large documents, creating an in-memory tree of the entire document requires significant memory resources. Second, the parse nodes on the tree do not become available until the entire document has been parsed.

# The Document Object Model

The Document Object Model is a way of representing well-formed XML documents as hierarchies of named objects, each of which can be manipulated in various ways. It is a public standard, managed by the W3C (<http://www.w3.org>), and is available in a number of programming languages, including TOOL.

## DOM Trees

As a DOM parser processes a document it creates in memory a tree structure representing the relationship of the objects in the document. The outermost object in the document becomes the root node of the tree. Each successive object is added as a subnode of the node within which it is nested. Branches are created when a node has more than one child node. Leaf nodes are those nodes at the end of a branch, that have no child nodes.

## DOM Tree Examples

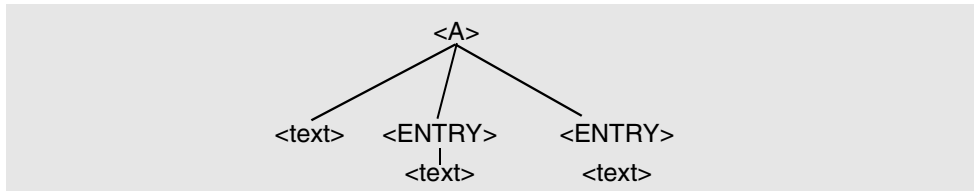
For example, consider the following XML document fragment:

### Code Example 20-1 Simple XML Sample

```
<A>
<ENTRY>
LAST_NAME="Appleseed"
FIRST_NAME="Johnny"
ADDRESS="1417 Gravenstein Way"
CITY="Oakland"
STATE="California"
ZIPCODE="94612"
PHONE="510-111-2222"
EMAIL="jappleseed@orchard.org"
</ENTRY>
<ENTRY>
LAST_NAME="Arthur"
FIRST_NAME="Melvin"
ADDRESS="921 Boulevard St."
CITY="Tehachapi"
STATE="California"
ZIPCODE="91218"
PHONE="432-999-1111"
EMAIL=NIL
</ENTRY>
</A>
```

When parsed, this fragment would produce a node tree that looked like this:

**Figure 20-1** Simple XML DOM Tree



In this tree the root node is "A." Its first child is a text node, and its second and third are both "ENTRY" elements. Each of these ENTRY elements contains text.

Note the first text node (the leftmost node under A). Such text nodes occur when whitespace exists in a document between elements of actual content. Typically they are spaces, tabs, or carriage return/line feeds. While it is easy to forget or ignore these text nodes, be aware that the XML parser treats them as being significant.

## Creating DOM Trees

A DOM tree can be created either by importing an existing text document into the DOM with the `Document.ImportDocument()` method, or it can be built by invoking various methods for creating particular objects. The following table identifies methods for creating objects.

**Table 20-1** Methods for Creating Objects

Object	Method
Attribute	<code>Document.createAttribute()</code>
Attribute with namespace	<code>Document.createAttributeNS()</code>
CDATASection	<code>Document.createCDATASection()</code>
Comment	<code>Document.createComment()</code>
Document	<code>DOMImplementation.createDocument()</code>
DocumentFragment	<code>Document.createDocumentFragment()</code>
DocumentType	<code>DOMImplementation.createDocumentType()</code>
Element	<code>Document.createElement()</code>
Element with namespace	<code>Document.createElementNS()</code>
EntityReference	<code>Document.createEntityReference()</code>
ProcessingInstruction	<code>Document.createProcessingInstruction()</code>
TextNode	<code>Document.createTextNode()</code>

For example, you could invoke the `Document.createElement()` method to create a new element called "ENTRY" and add it as a child of the element "root":

```
e:Element = doc.createElement(tagName='ENTRY')
root.appendChild(e);
```

## Reading DOM Trees

All objects in a DOM tree are accessible by using the appropriate `GET` method. The following methods fetch objects at various points in the tree:

**Table 20-2** Methods for Fetching Objects

Method	What it fetches
<code>Node.getChildNodes()</code>	Returns a <code>NodeList</code> containing all of the children of this node. If there are no children, this is a <code>NodeList</code> containing no nodes.
<code>Node.getFirstChild()</code>	Returns the first child of this node. If there is no such node, this returns <code>NIL</code> .
<code>Node.getLastChild()</code>	Returns the last child of this node. If there is no such node, this returns <code>NIL</code> . If the node has only one child <code>getLastChild()</code> returns the same result as <code>getFirstChild()</code> .
<code>Node.getNextSibling()</code>	Returns the node immediately following this node. If there is no such node, this returns <code>NIL</code> .
<code>Node.getParentNode()</code>	Returns the parent of this node. All nodes, except <code>Attr</code> , <code>Document</code> , <code>DocumentFragment</code> , <code>Entity</code> , and <code>Notation</code> may have a parent. However, if a node has just been created and not yet added to the tree, or if it has been removed from the tree, <code>getParentNode</code> returns <code>NIL</code> .
<code>Node.getPreviousSibling()</code>	Returns the node immediately preceding this node. If there is no such node, this returns <code>NIL</code> .

## Manipulating DOM Trees

The objects in a DOM tree can also be modified, using various methods. Elements can have their values set and modified, and in some cases can be replaced with other elements. The following table summarizes some of the methods that can be used to modify objects in a DOM tree.

**Table 20-3** Methods for modifying objects or their contents

Object	Method
Attribute	<code>Attr.setValue()</code>
CDATASection	<code>Node.setNodeValue()</code>
Comment	<code>CharacterData.replaceData()</code> , <code>CharacterData.setData()</code>
Document	<code>Node.setNodeValue()</code>

**Table 20-3** Methods for modifying objects or their contents (*Continued*)

<b>Object</b>	<b>Method</b>
DocumentFragment	Node.setNodeValue()
Element	Node.setNodeValue()
EntityReference	Node.setNodeValue()
NamedNodeMap	NamedNodeMap.removeNamedItem(), NamedNodeMap.setNamedItem()
ProcessingInstruction	ProcessingInstruction.setData(), ProcessingInstruction.setNodeValue()
TextNode	CharacterData.replaceData(), CharacterData.setData(), Text.splitText(), Normalize()

Nodes of all types can be added, removed, and copied with the following methods:

**Table 20-4** Methods for adding, removing, and copying nodes

<b>Method</b>	<b>Action</b>
Node.appendChild()	Adds node as a child of specified node
Node.insertBefore()	Adds node above the specified node
Node.removeChild()	Removes a child of the specified node
Node.replaceChild()	Replaces the specified child node with another node
Node.cloneNode()	Copies a node

### *Text Nodes*

Text data can be fetched by range, by specifying a byte offset at which the range begins and a length (in bytes) from the offset, using the `CharacterData.substringData()` method. Similarly, text data can be inserted into a text node by invoking the `CharacterData.insertData()` method. Text nodes can be divided using the `Text.splitText()` method, thus creating two siblings from a single node; they can also be combined, using the `Node.normalize()` method.

Like `TextData` objects, text node offsets are byte offsets, and start from 0. This is important when working with multi-byte character sets.

## The DOM API Classes

The following table summarizes the DOM API classes.

**Table 20-5** The DOM API classes

<b>Class or Interface</b>	<b>Description</b>
Attr	Represents an attribute in an Element object.
CDATASection	Used to escape blocks of text containing characters that would otherwise be parsed as markup tags.
CharacterData	Abstract base class for tree node classes containing character data from the document.
Comment	Represents the content of a comment in an XML document.
Document	Represents the entire XML document. It is the “root” of the document tree, and provides access to the document’s data and type information.
DocumentFragment	Provides a “lightweight” object that can be used to store sections of a document.
DocumentType	Provides a means of reading a list of entities defined for the particular document type of the current document.
DOMException	DOM methods raise exceptions of this class when error conditions are encountered.
DOMImplementation	Provides methods for performing operations that are independent of any particular instance of a document.
Element	Represents an element in an XML document.
Entity	Represents a parsed or unparsed entity in an XML document.
EntityReference	Represents an entity within or outside of the current document. Note that the XML parser may expand entity references when creating the document tree.
NamedNodeMap	Used to represent collections of nodes that can be accessed by name, but that have no inherent order.
Node	Node is the primary datatype of the Document Object Model, and is the base class for all document tree node classes.
NodeList	Provides an ordered collection of nodes.
Notation	Represents a notation declared in the DTD.
ProcessingInstruction	Represents a “processing instruction” in an XML document.
Text	Represents the textual content of an Element or Attribute.



## DOM Level 2 Features

DOM Level 2 provides a number of enhancements over Level 1.

### *Support for Namespaces*

The iPlanet UDS XMLDOM2 library includes a number of methods for creating and referencing nodes with namespace prefixes prepended to their URIs. This makes use of the XML namespaces more straightforward than in DOM Level 1, since the application need not parse qualified names, nor manage prefix bindings.

### *Improved Exception Handling*

DOM Level 2 provides 5 new exception codes, each of which represent a condition causing an operation to fail. The following table summarizes the new exception codes for DOM Level 2.

**Table 20-6** New Exception Codes for DOM Level 2

Exception Code	Exception
INVALID_ACCESS_ERR	A parameter or an operation is not supported by the object.
INVALID_MODIFICATION_ERR	An attempt is made to modify the type of the object.
INVALID_STATE_ERR	An attempt is made to use an object that is not, or is no longer, usable.
NAMESPACE_ERR	An attempt is made to create or change an object that is incorrect in terms of namespaces.
SYNTAX_ERR	An invalid or illegal string is specified.

### *DOM Object Attributes Available as TOOL Virtual Attributes*

The W3C specification for DOM Level 2 includes attributes for the various class types, which are implemented in the XMLDOM2 library as TOOL virtual attributes. These attributes were not available in the XMLParser library DOM Level 1 implementation.

### *Strings as Parameters to Text Functions and as TextData Types*

The DOM Level 2 specification stipulates that applications must encode DOMString data using the UTF-16 character set. In the XMLDOM2 library the DOMString datatype is mapped to the TOOL string type, and the TextData type is allowed as appropriate. Offset values and lengths are specified in bytes, exactly as they are in the TextData class.

## Upgrading from DOM Level 1

The iPlanet UDS XMLDOM2 library supports backwards compatibility for applications based on DOM Level 1. The XMLParser library (the UDS implementation of DOM Level 1) is still included in UDS, thus providing complete compatibility with existing applications.

However, be aware that in DOM Level 2 some parameter names and types have changed, and this may require modification of application code in order to switch from XMLParser to XMLDOM2.

In addition, if you want to make use of the namespace support included in DOM Level 2 you will need to review your current application code and rename elements and attributes to include namespace data.

# Using the XMLSAX2 Library

The Simple API for XML (SAX) is a library that provides a programmatic way of reading well-formed XML documents. SAX parsers convert XML documents into a sequence of parse events, for which user-supplied handlers can register.

SAX is not a public standard. It was developed from the input of a group of XML developers, and is managed by Dave Megginson. While it does not have the same official status as the various libraries standardized by the W3C organization, SAX has become a de facto standard among XML developers.

The iPlanet UDS XMLSAX2 class library supports the core features and properties of SAX Level 2, as defined at <http://www.megginson.com/SAX/Java/index.html>.

This chapter includes sections on the following topics.

- XML Namespaces, on [page 604](#)
- SAX and the Event Handling Model, on [page 604](#)
- Exception Handling in SAX Level 2, on [page 610](#)
- New Features in SAX Level 2, on [page 610](#)
- Upgrading from SAX Level 1, on [page 611](#)
- The XMLSAX2 classes, on [page 611](#)

## XML Namespaces

XML namespaces are used to qualify attribute and element names in XML documents, by associating them with URI references that identify unique namespaces. This allows applications to distinguish similarly named elements and attributes from different XML vocabularies, even if they are mixed together in the same document. These names are shortened by using prefixes. Prefixes are bound to namespaces by using the `xmlns` attribute.

A prefixed name is referred to as a *qualified name*. The two components of a qualified name are the prefix and the local name. For example:

```
<xsl:output xmlns:xsl="http://www.w3.org/1999/XSL/Transform" />
```

In this example, the first part “xsl” is the prefix, and the second part “output” is the local name. Together they form the qualified name.

When SAX elements and attributes are created they are permanently associated with a namespace URI.

Neither the validity of a namespace, nor the mapping between namespace prefixes and URIs, is checked by SAX. If required, these functions must be implemented at the application level.

## SAX and the Event Handling Model

At present there are two distinct models for parsing XML documents. In *tree-based parsing*, as implemented with the Document Object Model (DOM), the parser scans the XML document and creates an in-memory tree representing the document elements and their relationships to each other. The main advantage of tree-based parsing is that the resulting tree represents the entire document, and once the tree has been constructed every parse node on the tree is immediately available. For applications where specific nodes are manipulated tree-based parsing is very efficient. However, tree-based parsing uses large amounts of memory, and does not make the document available until the entire tree has been created.

The other model for parsing XML documents is *event-based parsing*, as implemented with the SAX library. In event-based parsing the parser scans the XML document and generates *events* for each syntactic element it finds. Applications can register event handlers for these events, and can initiate any sort of processing they wish on receiving events.

There are several advantages to event-based parsing. First, the application does not have to wait for the parser to scan the entire document before receiving output from the parser; events are generated immediately when the parser encounters a syntactic element. The application can begin processing the document immediately, without having to wait for the entire document tree to be generated. Second, this model is very quick and efficient in searching for a particular element in a document: the search concludes as soon as the parser has located the desired element, and no further scanning need be done. Third, event-based parsing can be very efficient in terms of memory use, especially if the application does not build an in-memory representation of the entire document.

## What Are Events?

It is important to note that the term *event*, as used herein and in other documents relating to the SAX library, has a different meaning than that term has in the TOOL language.

As used in SAX, the term *event* refers to SAX callback methods. In this context *event* is completely separate from the use of the term to refer to the TOOL event handler type.

An event occurs when the SAX parser encounters a syntactic element while scanning an XML document. A syntactic element is anything that is meaningful within the context of XML. This includes XML tags, actual content, and whitespace (spaces, tabs, carriage return/line feeds). A registered event handler is called by the parser when each XML element is encountered.

## Examples of Events

Consider the following simple XML document:

### Code Example 21-1 Simple XML Document

```
<?xml version="1.0"?>

<doc xmlns:co="http://songs.ca/traditional">
<co:verse>The sun was setting in the west
The birds were singing on every tree
All nature seemed inclined for to rest
But still there was no rest for me.</co:verse>
<co:chorus>Farewell to Nova Scotia, you sea-bound coast
Let your mountains dark and dreary be
For when I am far away on the briny ocean tossed
Will you ever heave a sigh and a wish for me?</co:chorus>
<co:verse>I grieve to leave my native land
I grieve to leave my comrades all
And my parents whom I held so dear
And the bonnie, bonnie lassie that I do adore.</co:verse>
<co:verse>The drums they do beat and the wars to alarm
The captain calls, we must obey
So farewell, farewell to Nova Scotia's charms
For it's early in the morning I am far, far away.</co:verse>
<co:verse>I have three brothers and they are at rest
Their arms are folded on their breast
But a poor simple sailor just like me
Must be tossed and driven on the dark blue sea. </co:verse></doc>
```

As the SAX parser begins scanning this document the first event it encounters is the beginning of the document. The first syntactic element it encounters is a “start document” tag. Immediately following that is a text element whose content is a carriage return. Following the carriage return is a “start form” tag.

Note that, because SAX Level 2 is namespace-aware, each element is uniquely identified with its namespace.

When the parser has finished scanning the document it will have generated the following series of parse events:

### Code Example 21-2 Output generated by parsing simple XML document

```
StartDocument ()
StartElement ( " " "doc" "doc" )
Characters ( "\n" )
StartElement ( "http://songs.ca/traditional" "verse" "co:verse" )
```

**Code Example 21-2** Output generated by parsing simple XML document (*Continued*)

```

Characters( "The sun was setting in the west" )
Characters( "\n" )
Characters( "The birds were singing on every tree" )
Characters( "\n" )
Characters( "All nature seemed inclined for to rest" )
Characters( "\n" )
Characters( "But still there was no rest for me." )
EndElement( "http://songs.ca/traditional" "verse" "co:verse" )
Characters( "\n" )
StartElement( "http://songs.ca/traditional" "chorus" "co:chorus" )
Characters( "Farewell to Nova Scotia, you sea-bound coast" )
Characters( "\n" )
Characters( "Let your mountains dark and dreary be" )
Characters( "\n" )
Characters( "For when I am far away on the briny ocean tossed" )
Characters( "\n" )
Characters( "Will you ever heave a sigh and a wish for me?" )
EndElement( "http://songs.ca/traditional" "chorus" "co:chorus" )
Characters( "\n" )
StartElement( "http://songs.ca/traditional" "verse" "co:verse" )
Characters( "I grieve to leave my native land" )
Characters( "\n" )
Characters( "I grieve to leave my comrades all" )
Characters( "\n" )
Characters( "And my parents whom I held so dear" )
Characters( "\n" )
Characters( "And the bonnie, bonnie lassie that I do adore." )
EndElement( "http://songs.ca/traditional" "verse" "co:verse" )
Characters( "\n" )
StartElement( "http://songs.ca/traditional" "verse" "co:verse" )
Characters( "The drums they do beat and the wars to alarm" )
Characters( "\n" )
Characters( "The captain calls, we must obey" )
Characters( "\n" )
Characters( "So farewell, farewell to Nova Scotia's charms" )
Characters( "\n" )
Characters( "For it's early in the morning I am far, far away." )
EndElement( "http://songs.ca/traditional" "verse" "co:verse" )
Characters( "\n" )
StartElement( "http://songs.ca/traditional" "verse" "co:verse" )
Characters( "I have three brothers and they are at rest" )
Characters( "\n" )
Characters( "Their arms are folded on their breast" )
Characters( "\n" )
Characters( "But a poor simple sailor just like me" )
Characters( "\n" )
Characters( "Must be tossed and driven on the dark blue sea. " )
EndElement( "http://songs.ca/traditional" "verse" "co:verse" )
EndElement( "" "doc" "doc" )
EndDocument()

```

## StartDocument() and EndDocument() Methods

At a minimum every well-formed XML document will generate two events: `START DOCUMENT` and `END DOCUMENT`. These events are unique, in that they can occur only once in a given document, and they take no arguments.

The application responds to `START DOCUMENT` events by invoking the registered `ContentHandler`'s `StartDocument()` method

`END DOCUMENT` events are processed with the registered content handler's `EndDocument()` method.

---

**NOTE** The `DefaultHandler` class is a convenience for developers creating content handlers. The `DefaultHandler` methods do nothing, and can be overridden to implement whatever functionality the developer wants.

---

## StartElement() andEndElement() Methods

The next most significant type of events are those that signal the beginning and end of elements. `START ELEMENT` events are handled with the registered `ContentHandler`'s `StartElement()` method, which takes three input parameters for identifying the element: `namespaceURI`, `localName`, and `qName`. `END ELEMENT` events are handled with the registered `ContentHandler`'s `EndElement()` method, which takes the same input parameters as `StartElement()`.

The `StartElement()` event is followed by events for all of the element's contents, in the order in which they occur in the document. Applications can respond by processing the contents of the element in whatever way they chose.

## Character Events

When the SAX parser encounters character data it generates a character event, which is processed by invoking the registered `ContentHandler`'s `Characters()` method.



# Filtering Events

Depending on the nature of your application and the XML documents it uses, it may be useful to filter events as they are passed from the parser to the application.

A possible use for a filter might be for type and range checking of attribute values. For example, suppose an XML document includes an attribute for US-format social security number. You might want to have a filter that checks to make sure the attribute value has nine digits, and that it is appropriately formatted as three digits, then two digits, then four digits. Or you might want to have a filter that checks to make sure a particular attribute contains only numeric data.

Filters can also be useful during the development and testing of your application. For example, you might want to have a filter that simply logs all events generated by a particular document.

Filters can be arranged in chains with the `FilterImpl.SetContentHandler()` method, allowing for flexible composition of event filters at runtime.

Filters are connected to XML readers by invoking the `FilterImpl.SetParent()` method.

## The FilterImpl Class

The `FilterImpl` class is a base class from which application developers can build application-specific filters. It includes methods for filtering:

- `START DOCUMENT` and `END DOCUMENT` events
- `START ELEMENT` and `END ELEMENT` events
- character events
- ignorable whitespace events
- error events
- entity resolution events
- document locator events
- skipped entity events
- namespace prefix mapping events
- unparsed entity declaration events

Each of these methods can be subclassed to provide application-specific filtering.

FilterImpl also includes methods for setting event handlers, parsers, DTD handlers, and error handlers.

The default behavior of all FilterImpl methods is to convey requests to the parent, and convey all events to the registered handlers without modification.

## Exception Handling in SAX Level 2

The default SAX Level 2 exception handling is implemented with the ErrorHandler interface. If no error handler has been registered, the reader raises the exception if it is fatal. XML documents that are not well-formed raise fatal errors.

SAXParseException includes methods for locating errors within the XML document.

Alternatively, developers can create application-specific exception handling mechanisms by implementing the ErrorHandler interface, and then registering an instance with the parser, by using the XMLReader.setErrorHandler() method.

## New Features in SAX Level 2

SAX Level 2 adds several new features.

### Support for Namespaces

The XMLReader interface (which SAX parsers implement) includes support for namespace processing. This applies to elements and attributes, which in SAX Level 1 were identified only by a qualified name. With the XMLSAX2 library, all of the ContentHandler methods are capable of processing elements and attributes identified by the combination of a URI, local name and qualified name.

### Configurable Parsers

SAX Level 2 parsers are configurable via the XMLReader.setFeature() method, which enables the developer to specify a feature (by namespace) and turn that feature on for a particular parser. An example of such a feature is XML validation. If a particular feature is not supported by the parser a SAXNotSupportedException is thrown.

## Other Features

Skipped entity events can now be reported, via the `ContentHandler.skippedEntity()` method. This is useful, because parsers may skip external entities or entities that were declared in an external DTD.

The `Attributes` interface enables access to attributes by index value, and adds support for attribute namespaces.

## Upgrading from SAX Level 1

The iPlanet UDS includes both the `XMLParser` and `XMLSAX2` libraries, and thus SAX Level 1 applications remain fully supported.

Developers wanting to make use of the SAX Level 2 features need to be aware that Level 1 applications will need to be modified to support namespaces. Additionally, some parameter names and types have changed, and existing applications may need to be modified accordingly. In particular, `DocumentHandler` implementations must be modified to become `ContentHandler` implementations. In addition, SAX Level 2 uses `XMLReaders`, instead of `Parsers`.

## The XMLSAX2 Classes and Interfaces

The following table summarizes the classes and interfaces that make up the XMLSAX2 library. (Note that interface names are italicized.)

**Table 21-1** The XMLSAX2 classes and interfaces

<b>Class or Interface</b>	<b>Description</b>
<i>AttributeList</i>	Original SAX Level 1 interface for reporting an element's attributes. Does not support Namespace-related information. Now deprecated. Included for support of SAX Level 1 applications.
<i>Attributes</i>	Allows access to a list of attributes, by index, namespace, or local name.

**Table 21-1** The XMLSAX2 classes and interfaces (*Continued*)

<b>Class or Interface</b>	<b>Description</b>
<i>ContentHandler</i>	Main interface for implementing content handlers. If the application needs to be informed of basic parsing events, it implements the ContentHandler interface and registers an instance with the SAX parser using the setContentHandler() method. The parser uses the instance to report basic document-related events like the start and end of elements and character data.
DefaultHandler	Convenience base class for SAX2 applications. Provides default implementations for all callbacks in the four core SAX2 handler classes: EntityResolver, DTDHandler, ContentHandler, and ErrorHandler.
<i>DocumentHandler</i>	Main event-handling interface for SAX1. Now deprecated. Has been replaced by ContentHandler, which provides Namespace support and reporting of skipped entities. Included for support of SAX Level 1 applications.
<i>DTDHandler</i>	Used to receive notification of basic DTD-related events. If an application needs information about notations and unparsed entities the application implements DTDHandler and registers an instance with the SAX parser using the parser's setDTDHandler() method. The parser uses the instance to report notation and unparsed entity declarations to the application.
<i>EntityResolver</i>	Basic interface for resolving entities. If an application needs to implement customized handling for external entities, it must implement this interface and register an instance with the SAX driver using the setEntityResolver() method. The XML reader will then allow the application to intercept any external entities (including the external DTD subset and external parameter entities, if any) before including them.
<i>ErrorHandler</i>	Basic interface for SAX error handlers. If an application needs to implement customized error handling, it must implement this interface and then register an instance with the XML reader using the setErrorHandler() method. The parser will then report all errors and warnings through this interface.
HandlerBase	Default base class for SAX Level 1 handlers. Now deprecated. Replaced with the DefaultHandler class. Included for support of SAX Level 1 applications.

**Table 21-1** The XMLSAX2 classes and interfaces (*Continued*)

<b>Class or Interface</b>	<b>Description</b>
InputSource	Single input source for an XML entity. Allows application to encapsulate information about an input source in a single object, which may include a public identifier, a system identifier, and/or a character stream.
<i>Locator</i>	Used for associating a SAX event with a document location. If the parser provides location information to the application, it does so by implementing this interface and then passing an instance to the application using the content handler's <code>setDocumentLocator()</code> method. The application can use the object to obtain the location of any other content handler event in the XML source document.
LocatorImpl	Convenience implementation of <code>Locator</code> . Available for application writers, who can use it to make a persistent snapshot of a locator at any point during a document parse.
<i>Parser</i>	Basic interface for SAX Level 1 parsers. Now deprecated. Replaced in SAX Level 2 by <code>XMLReader</code> , which includes namespace support and greater configurability and extensibility.
ParserFactory	Class for creating SAX Level 1 parsers. Now deprecated. Replaced by <code>XMLReaderFactory</code> . Included for support of Level 1 applications.
SAXException	Contains basic error or warning information from either the XML parser or the application. Can be subclassed to provide additional functionality. SAX handlers may throw this exception or any exception subclassed from it. If the application needs to pass through other types of exceptions, it must wrap those exceptions in a <code>SAXException</code> or an exception derived from a <code>SAXException</code> .
SAXNotRecognizedException	Exception class for an unrecognized identifier. An <code>XMLReader</code> throws this exception when it finds an unrecognized feature or property identifier.
SAXNotSupportedException	Exception class for an unsupported operation. An <code>XMLReader</code> throws this exception when it recognizes a feature or property identifier, but cannot perform the requested operation (setting a state or value).

**Table 21-1** The XMLSAX2 classes and interfaces (*Continued*)

<b>Class or Interface</b>	<b>Description</b>
SAXParseException	Encapsulates an XML parse error or warning. Includes information for locating the error in the original XML document.
<i>XMLFilter</i>	Interface for an XML filter. Filters can modify a stream of events as they pass on to the final application.
XMLFilterImpl	Base class for deriving an XML filter. Designed to sit between an XMLReader and the client application's event handlers. By default, it does nothing but pass requests up to the reader and events on to the handlers unmodified, but subclasses can override specific methods to modify the event stream or the configuration requests as they pass through.
<i>XMLReader</i>	Interface for reading an XML document using callbacks. Allows an application to set and query features and properties in the parser, to register event handlers for document processing, and to initiate a document parse.
XMLReaderFactory	Factory for creating an XML reader. Contains methods for creating an XML reader from an explicit class name, or for creating an XML reader from a library.

# Accessing Internet Directory Services

iPlanet UDS allows you to access to Internet directory services available from a server implementing the Lightweight Directory Access Protocol (LDAP). This chapter provides a brief overview of the LDAP protocol, and then describes how to create an iPlanet UDS application that connects to and queries an LDAP server.

This chapter covers the following key topics:

- “LDAP Overview” on page 615
  - “Accessing and Updating an LDAP Directory” on page 616
- “Using the iPlanet UDS LDAP Library” on page 617
  - “Establishing an LDAP Session” on page 617
  - “Searching an LDAP Directory” on page 619
  - “Building LDAP Filters” on page 621
  - “Updating an LDAP Directory” on page 623

## LDAP Overview

The Lightweight Directory Access Protocol (LDAP) is a specification for access to directory services over the Internet. An LDAP directory service stores information as entries in a directory that is implemented as an hierarchical tree with specific naming formats. You can query an LDAP directory to retrieve specific information and you can also update the information in the directory.

## LDAP Directory Information

Information in an LDAP directory is stored as *entries* that describe some real-world object (for example, a person, an organization, or a node on a network). Each entry contains *attributes* that provide information about the object. Each attribute has a *type* that describes the kind of information the attribute holds. An attribute can be associated with one or more *values*.

Examples of attribute types include *cn* (for common name), *mail* (for email address), and *telephonenumber* (for telephone number). Each type defines a syntax that defines how to store the values for each type. For example, the syntax of the *cn* type specifies that the values must be character strings and that case is ignored during comparisons.

## LDAP Directory Trees

Entries in an LDAP directory are stored as hierarchical trees—typically the top level of the tree represents countries, followed by states and then organizations. Some LDAP directory trees use domain names at the top level, and then branch into organizations.

Entries use a series of one or more attribute/value pairs to form a *relative distinguished name* (RDN), which must be unique among its siblings. An entry's *distinguished name* (DN) is the complete series of RDNs from the entry to the top level of the tree. An entry's DN is unique within the tree, and thus allows you to reference a specific entry.

A distinguished name is represented as a set of RDN components separated by commas ',' or semi-colons ';'. For example, The following distinguished name references an employee in an organization:

```
cn=Raggs Ginsberg, ou=Marketing, o=iPlanet, c=US
```

## Accessing and Updating an LDAP Directory

Typically, a client to an LDAP directory server obtains directory information by specifying a search constrained by a set of specified filters. The search query returns matching entries with a requested set of attributes and their values. The client can constrain the search according to the number of entries found and also specify a timeout period for the search.

A client can perform the following operations to update an LDAP directory:



- **Modify**  
Change the attributes and values of an entry.
- **Add/Delete**  
Insert or remove entries from the directory.

## Using the iPlanet UDS LDAP Library

The iPlanet UDS LDAP library allows you to connect to and query an LDAP directory server for information. It also provides entry points that allow you to update entries in an LDAP directory. To access an LDAP directory server you perform the following steps:

1. Establish an LDAP session.

Refer to the following section, [“Establishing an LDAP Session”](#) for information on establishing sessions.

2. Perform a directory service operation.

Typically, you query for information using filters, as described in [“Searching an LDAP Directory”](#) on page 619.

However, you can also update an entry by adding, deleting, or modifying its attributes and values. Refer to [“Updating an LDAP Directory”](#) on page 623 for more information.

3. Close the LDAP session.

Refer to [“Closing an LDAP Session”](#) on page 628 for information on closing a session.

## Establishing an LDAP Session

Before an iPlanet UDS application connects to an LDAP server, it must first create an LDAP session. The session then connects to the LDAP server by specifying the following information about the address of the server:

- LDAP server URL
- Port number
- Type of address

## Connecting to an LDAP Server

**Code Example 22-1** shows how to use `LDAPSession.Open` to connect to an LDAP server.

### Code Example 22-1 Connecting to an LDAP server

```
-- open a connection to machine dir.iPlanet.com
aLDAP : LDAPSession = new();
aLDAP.Open( pAddress = textdata( 'dir.iPlanet.com' ),
            pPort=389, pAddressType = CM_ADDR_INTERNET_NAME );
```

## Message IDs

After establishing a session with an LDAP server, each request to the server must contain a unique message ID. Typically, the client application increments a counter to guarantee unique message IDs for each request.

## Authentication

After connecting to the server, the client application authenticates itself to the server using the `LDAPSession.Bind` method.

---

**NOTE** Many LDAP servers do not enforce the bind authentication requirement.

---

**Code Example 22-2** provides an example of binding to an LDAP server. When binding to the server, you specify a unique message ID, the name of the directory object you are binding as, and a password (if required by the server).

### Code Example 22-2 Authenticating a session with the LDAP server

```
-- bind as Directory Manager
messageID : integer = 1;
binddn : Textdata( value = 'cn=Directory Manager' );
passwd : Textdata( value = 'dirmanager' );

status : TextData;
status = aLDAP.Bind( messageID, binddn, password );
```

## Searching an LDAP Directory

A search of an LDAP directory specifies various criteria, including a filter to constrain your search, where to begin your search, the scope of the search, how much data to return, a list of attributes to use for comparison, and an object to hold the results of the search.

The `LDAPSession.Search` method has two signatures. One signature specifies a `TextData` parameter to hold the result. The other specifies an array of `LDAPEntry` for the result. The syntax for the first signature is:

```
Search(Integer messageID, TextData base, Integer scope,
        Integer aliases, Integer sizeLimit, Integer timeLimit
        Boolean typesOnly, LDAPFilter filter,
        Array of TextData attributes, TextData result)
```

**Table 22-1** summarizes the type of criteria you specify as parameters to `Search`. **Code Example 22-3** on page 620 illustrates a typical example of using the `Search` method.

For more information on the `Search` method, refer to the online help.

**Table 22-1** LDAPSession.Search Parameters

Search Parameter	UDS Datatype	Description
Message ID	Integer	The unique ID used to establish a connection with the LDAP server.
Search Base	TextData	The starting point for the search. You can specify the base node of the LDAP directory tree or a specific node in the tree.
Scope	Integer	You can specify the entire LDAP directory, a specific node, or a specific node and the child nodes.
Aliases	Integer	Dereferencing aliases is currently not supported. Specify zero to indicate <i>do not dereference aliases</i> .
Size Limit	Integer	The maximum number of entries to return for the search. A specification of zero means no limit.
Time Limit	Integer	The maximum number of seconds allowed for the search. A specification of zero means no time limit.
Types Only	Boolean	TRUE specifies to return just the attribute types (no values). FALSE specifies to return the attribute types with values.

**Table 22-1** LDAPSession.Search Parameters (Continued)

Search Parameter	UDS Datatype	Description
Filter	LDAPFilter	Specifies the conditions that must be fulfilled to find matching entries in the directory.
Attributes	Array of TextData	A list of attributes to be returned for matching entries.
Result	TextData	The result of the search, returned as text.
Result Array	Array of LDAPEntry	The result of the search, returned as an array of LDAP entries.

**Code Example 22-3** provides an example of an LDAP directory search that looks for entries in an organization where the common name (cn) begins with 'R', ends with 'G', and contains 'A'. The search returns the common name and simple name for matching entries as TextData. This search is designed to return entries that would include the entry for the employee named "Raggs Ginsberg."

This example builds a filter to search for the individual employee. For more information on LDAP filters, refer to "[Building LDAP Filters](#)" on page 621.

**Code Example 22-3** Specifying an LDAP directory search

```
-- Create a new Message ID (increment the previous Message ID)
messageID = messageID + 1;

-- Search base begins at the specified organization
aBase : TextData( 'o=iPlanet.com' );

-- Search includes the search base and its immediate children
aScope : Integer = LDAP_SCOPE_WHOLE;

-- Build a search filter that specifies r*a*g for cn
aFilter : LDAPFilter;
aFilter.Type = LDAP_FILTER_SUBSTR;
aFilter.attributeValueDesc = 'cn';
aFilter.Initial = 'r';
aFilter.any = 'a';
aFilter.final = 'g';

-- Return the cn and sn attributes
theAttributeList : Array of TextData = new();
theAttributeList[1] = TextData( 'cn' );
theAttributeList[2] = TextData( 'sn' );

-- Specify the search:
-- Do not search any aliases
```

**Code Example 22-3** Specifying an LDAP directory search (*Continued*)

```

-- Do not limit the number of entries
-- Do not limit how long the search takes
-- Return attributes with their values
-- Return the result in a TextData object
-- Note: reuse the status TextData object defined earlier
TextData : Result;
status = aLDAP.Search( MessageID, aBase, aScope,
                      0, 0, 0, FALSE, aFilter,
                      theAttributeList, Result);

```

**Building LDAP Filters**

The iPlanet UDS LDAP library provides several types of filters you can use in an LDAP directory search. This section summarizes the type of filters you can create. For more information on the types of LDAP filters, refer to the online help.

---

**NOTE** Typically, an LDAP server ignores case during searches.

---

*Equal Filter*

The filter type LDAP\_FILTER\_EQUAL specifies a string for an exact match. For example, to search for the cn attribute value “Cosmo” (cn=cosmo), do the following:

**Code Example 22-4** Building an “equals” filter

```

filter : LDAPFilter = new();
filter.type = LDAP_FILTER_EQUAL;
filter.AssertionValue( 'Cosmo' );
filter.attributeValueDesc.SetValue( 'cn' );

```

*Greater and Less Than Filters*

Two filter types specify strings that are lexicographically greater than or less than a specified string.

LDAP\_FILTER\_GEQ specifies strings greater than or equal to the specified string; LDAP\_FILTER\_LEQ specifies strings less than or equal to the specified string.

For example, to search for cn attribute values that are greater than “Cosmo” (cn>=cosmo), do the following:

**Code Example 22-5** Building a “lexicographic greater than” filter

```
filter : LDAPFilter = new();
filter.type = LDAP_FILTER_GEQ;
filter.AssertionValue( 'Cosmo' );
filter.attributeValueDesc.SetValue( 'cn' );
```

### *Substring Filter*

The filter type LDAP\_FILTER\_SUBSTR specifies a substring of an LDAP entry’s attribute value. For example, to search for a cn attribute value that begins with ‘J’ and contains the string “BERT” (cn=g\*funk\*), do the following:

**Code Example 22-6** Building a substring filter

```
filter : LDAPFilter = new();
filter.type = LDAP_FILTER_SUBSTR;
filter.initial.SetValue( 'g' );
filter.any.SetValue( 'funk' );

filter.attributeValueDesc.SetValue( 'cn' );
```

### *Approximate Filter*

The filter type LDAP\_FILTER\_APPROX specifies a string for an approximate match. For example, to search for a cn attribute value that is similar to “USIR” (cn~=USIR), do the following:

**Code Example 22-7** Building a substring filter

```
filter : LDAPFilter = new();
filter.type = LDAP_FILTER_APPROX;
filter.AssertionValue( 'usir' );
filter.attributeValueDesc.SetValue( 'cn' );
```

### *Attribute Present*

The LDAP\_FILTER\_PRESENT matches all entries that contains a specified attribute. For example, to search for all entries where the cn attribute is present, regardless of the value of the attribute, do the following:

#### **Code Example 22-8** Building an “any attribute value” filter

```
filter : LDAPFilter = new();
filter.type = LDAP_FILTER_PRESENT;
filter.attributeValueDesc.SetValue( 'cn' );
```

## Updating an LDAP Directory

The iPlanet UDS LDAP library supports the following operations to update an LDAP directory:

- Add an entry
- Modify an entry by adding, deleting, or modifying its attributes or attribute values
- Delete an Entry

When updating an LDAP directory, the LDAP server returns a response code that indicates the success (or failure) of the update request.

### Adding an LDAP Entry

To add an entry to an LDAP directory, create an instance of the LDAPEntry class and specify a distinguished name (DN) for the entry. You then add LDAP attributes and their values to an array of LDAPAttributes associated with the entry object. Finally, you call LDAPSession.Add to add the entry to the LDAP directory.

The syntax of LDAPSession.Add is:

```
Add(Integer messageID, LDAPEntry entry,
      TextData Result) : TextData
```

*messageID* is a unique identifier for the LDAP session.

*entry* is the entry you are adding

*result* is currently not used, but must be specified in the call.

**Code Example 22-9** shows how to add an entry representing a new organization employee.

**Code Example 22-9** Adding an entry to an LDAP directory

```
-- Add an entry for an employee with the following attributes
-- ou=people o=iPlanet.com uid=rg
-- cn=Raggs Ginsberg sn=Ginsberg givenName=Raggs

-- Create a new Message ID (increment the previous Message ID)
messageID = messageID + 1;

-- Create an entry and specify its distinguished name
entry : LDAPEntry = new();
entry.DN = 'uid=rg, ou=People, o=iPlanet.com';

-- Add LDAP attributes and their values to the entry.
entry.AttnList.AppendRow( LDAPAttribute() );
entry.AttnList[1].AttName = 'objectclass';
entry.AttnList[1].AttValueList.AppendRow( TextData( 'top' ) );
entry.AttnList[1].AttValueList.AppendRow(
    TextData( 'person' ) );
entry.AttnList[1].AttValueList.AppendRow( TextData(
    'organizationalPerson' ) );
entry.AttnList[1].AttValueList.AppendRow( TextData(
    'inetOrgPerson' ) );

entry.AttnList.AppendRow( LDAPAttribute() );
entry.AttnList[2].AttName='uid';
entry.AttnList[2].AttValueList.AppendRow( TextData( 'rg' ) );

entry.AttnList.AppendRow( LDAPAttribute() );
entry.AttnList[3].AttName = 'givenName';
entry.AttnList[3].AttValueList.AppendRow( TextData( 'Raggs' ) );

entry.AttnList.AppendRow( LDAPAttribute() );
entry.AttnList[4].AttName = 'sn';
entry.AttnList[4].AttValueList.AppendRow( TextData(
    'Ginsberg' ) );

entry.AttnList.AppendRow( LDAPAttribute() );
entry.AttnList[5].AttName = 'cn';
entry.AttnList[5].AttValueList.AppendRow( TextData(
    'Raggs Ginsberg' ) );

-- Update the LDAP directory. Note: reuse the aLDAP session and
-- status and Result TextData objects defined earlier
status = aLDAP.Add( MessageID, entry, Result );
```



## Modifying an Attribute for an LDAP Entry

To modify an attribute for an entry to an LDAP directory, create an instance of the `LDAPEntry` class. Then specify the distinguished name (DN) for the entry you want to modify and the type of operation you are making on the attribute of the entry. You then add the LDAP attributes and their values you want to modify to an array of `LDAPAttributes` associated with the entry object. Finally, you call `LDAPSession.Modify` to modify the attribute for the entry in the LDAP directory.

The syntax of `LDAPSession.Modify` is:

```
Modify(Integer messageID, Integer type,
        LDAPEntry entry, TextData Result) : TextData
```

*messageID* is a unique identifier for the LDAP session.

*type* see [Table 22-2](#)

*entry* is the entry you are modifying

*result* is currently not used, but must be specified in the call.

**Table 22-2** Modification Types

Type	Description
LDAP_MOD_ADD	Add values listed to the attribute in the given entry, creating the attribute if necessary.
LDAP_MOD_REPLACE	Replace all existing values of the attribute in the given entry with the new values listed, creating the attribute if it did not already exist. A replace with no value deletes the entire attribute if it exists, and is ignored if the attribute does not exist.
LDAP_MOD_DELETE	Delete values listed to the attribute in the given entry, removing the entire attribute if no values are listed, or if all current values of the attribute are listed for deletion.

[Code Example 22-10](#) shows how to modify the telephone number of a user in an organization. The example first adds the `phonenum` attribute, modifies its value, and deletes the `phonenum` attribute.

**Code Example 22-10** Modifying an LDAP directory entry

```

-- Add, Modify, then Delete the phonenumber attribute for entry:
-- uid=lb, ou=People, o=iPlanet.com

-- Create an entry and specify its distinguished name
entry : LDAPEntry = new();
entry.DN = 'uid=rg, ou=People, o=iPlanet.com';

-- ***** --
-- ADD the phonenumber attribute
Integer : Type = LDAP_MOD_ADD;

-- Create a new Message ID (increment the previous Message ID)
messageID = messageID + 1;

-- Add LDAP attributes and their values to the entry.
entry.AttList.AppendRow( LDAPAttribute() );
entry.AttList[1].AttName = 'phonenumber';
entry.AttList[1].AttValueList.AppendRow(
    TextData( '555-1212' ) );

-- Update the LDAP directory. Note: reuse the aLDAP session and
-- status and Result TextData objects defined earlier
status = aLDAP.Modify( messageID, Type, entry, result);

-- ***** --
-- Now, MODIFY the phonenumber attribute
Type = LDAP_MOD_REPLACE;
messageID = messageID + 1;

-- Change the phone number
entry.AttList[1].AttValueList[1].SetValue( '555-9999' );

-- Update the LDAP directory
status = aLDAP.Modify( messageID, Type, entry, result);

-- ***** --
-- Now, DELETE the phonenumber attribute
Type = LDAP_MOD_DELETE;
messageID = messageID + 1;

-- Delete the attribute values from the entry
entry.AttList[1].AttValueList.DeleteRow(1);

-- Update the LDAP directory
status = aLDAP.Modify( messageID, Type, entry, result);

```

## Deleting an LDAP Entry

To delete an entry from an LDAP directory, call `LDAPSession.Delete` with a `TextData` parameter specifying the distinguished name of the entry you are deleting.

The syntax of `LDAPSession.Delete` is:

```
Delete(Integer messageID, TextData dn) : TextData
```

*messageID* is a unique identifier for the LDAP session.

*dn* is the distinguished name of the entry you are deleting

**Code Example 22-11** shows how to delete an entry.

### **Code Example 22-11** Deleting an LDAP directory entry

```
-- delete the entry for:
-- uid=peterm, ou=People, o=iPlanet.com

-- Delete the LDAP entry.  Note: reuse the aLDAP session object
-- and the status TextData object defined earlier.
status = aLDAP.Delete( messageID,
                       'uid=peterm, ou=People, o=iPlanet');
```

## Closing an LDAP Session

Before closing an LDAP session, a client may want to “unbind” from the LDAP server. However, LDAP server implementations are not required to send a response to an unbind request. In the case where a server does not send a response, a call to `LDAPSession.Unbind` may hang as it waits for a response that never arrives.

### Code Example 22-12 Unbinding from an LDAP server

```
-- Create a new Message ID (increment the previous Message ID)
messageID = messageID + 1;

-- unbind as Directory Manager (reuse the status TextData object)
status = aLDAP.Unbind( MessageID );
```

To shut down an LDAP session, close the connection as illustrated in [Code Example 22-13](#).

### Code Example 22-13 Closing a session with the LDAP server

```
-- Close the connection
aLDAP.Close();
```

# iPlanet UDS Example Applications

iPlanet UDS provides a number of example applications that illustrate how to use TOOL and the iPlanet UDS classes. This appendix provides instructions on how to install the examples, a brief overview of the applications to help you locate relevant examples, and a section describing each example in detail. Typically, you run an example application, then examine it in the various iPlanet UDS Workshops to see how it is implemented. You can modify the examples if you wish.

## How to Install iPlanet UDS Example Applications

You can access the iPlanet UDS example applications only if they have been installed into your central repository or into a private local repository during installation of iPlanet UDS, or if you have imported them into your repository.

The examples are located in subdirectories under the FORTE\_ROOT/install/examples directory. The example applications are stored as .pex files. If they are not already installed in your repository, import them by including the tstapps.fsc script in Fscript. The tstapps.fsc script is located in the FORTE\_ROOT/install/examples/install directory. Bring up Fscript in standalone mode and issue the following command:

**Code Example A-1** Importing iPlanet UDS examples into a repository

```
fscript> UsePortable
fscript> SetPath %{\FORTE_ROOT}/install/examples/install
fscript> Include tstapps.fsc
```

This will import most of the example applications and quit Fscript. Note that certain highly specialized examples are not automatically imported by tstapps.fsc.

To run an application, select it in the Repository Workshop’s plan browser and then click on the Run button.

If you want to remove all the examples from your workspace, you can do so by including the remprj.fsc script in Fscript. Bring up Fscript in standalone mode and issue the following commands:

**Code Example A-2** Removing examples from a workspace

```
fscript> UsePortable
fscript> SetPath %{\FORTE_ROOT}/install/examples/install
fscript> Include remprj.fsc
```

This will exclude all the example applications and quit Fscript.

## Overview of iPlanet UDS Example Applications

This section provides an overview of the iPlanet UDS example applications. The following table lists the example applications that are referenced in this manual.

The margin note for each of the following tables shows the name of the subdirectory under FORTE\_ROOT/install/examples where you can find the .pex files for the examples. For the complete description of an individual application, see *“Application Descriptions” on page 631*, which lists the applications in alphabetical order.

	<b>Example</b>	<b>Description</b>
frame/	AdaptableAuction	Shows how to dynamically load an implementation of an interface.
frame/	AppletBanking	Illustrates launching of applets from a main application.
tool/	Auction	Illustrates prominent features of an iPlanet UDS distributed application.
display/	AutoTester	Shows how to use the Capture and Playback classes to automate GUI testing.
frame/	Banking1-2	Illustrates how to use convertors to run old and new clients and servers together.

	<b>Example</b>	<b>Description</b>
extsys/http	HTTP	Provides examples on implementing HTTP servers and HTTP clients.
display/	InheritedWindow	Shows how to subclass your own UserWindow classes.
internat/	InternatBank	Uses catalog files to translate windows and error messages.
display/	NestedWindow	Illustrates multiple tasks, nested windows, event handlers, and input validation.
frame/	NomadicOrderClient	Illustrates how to design an application that uses nomadic clients.
display/	PrintSample	Shows how to use the printing classes.
display/	TabFolders	Illustrates the use of the TabFolder class and Popup Menus.
frame/	TimeItV1-4	Illustrates runtime compatibility issues.
display/	TreeList	Shows how to coordinate data displayed in TreeView and List View fields.

## Application Descriptions

This section lists the example applications in alphabetical order. Each example has five sections describing it.

The **Description** section defines the purpose of the example, what problem it solves, and what TOOL features and iPlanet UDS classes it illustrates.

The **Pex Files** section gives you the subdirectory and file names of the exported projects. The examples are in subdirectories under the FORTE\_ROOT/install/examples directory. You can import example applications individually if you wish. When multiple .pex files are listed, there are supplier projects in addition to the main project. You will need to import all the files listed to run the application. Import them in the order given so that dependencies will be satisfied.

The **Mode** section indicates whether the application can be run in either standalone or distributed mode, or whether it must be run in distributed mode.

The **Special Requirements** section identifies whether you need a database connection, an external file, or any other special setup.

Finally, the **To Use** section tells you how to step through the application's functions.

See the *iPlanet UDS System Management Guide* if you need directions to set up an iPlanet UDS server. See *Accessing Databases* if you need information on how to make a connection to a database. The database examples run against either Sybase or Oracle.

## AdaptableAuction

**Description** AdaptableAuction shows how to use methods on the Partition and Library classes to dynamically load implementations of an interface.

A simple auction application calculates the taxes on the items sold. A tax calculation interface class provides the signature for a method which will calculate the tax on a sale item. Two implementations of this interface are provided. The implementations must be configured as libraries, in order to be dynamically loaded. The AdaptableAuction application reads from a file to determine at runtime which of the two implementations to load. You can copy over this file while the application is running and it will load the other implementation, which calculates the tax differently.

**Pex Files** frame/aa.pex

**Mode** Standalone or Distributed.

**Special Requirements** You will need to configure the two implementations as libraries and distribute them. You will need to copy a file while the application is running to see the dynamic loading behavior.

### ► To use Adaptable Auction

1. The two files dynload1.dat and dynload2.dat provide the library and class information used by the application at runtime. The application expects a file named dynload.dat to be in FORTE\_ROOT/install/examples/frame. Copy dynload1.dat to FORTE\_ROOT/install/examples/frame/dynload.dat.
2. Import the aa.pex file, which includes the interface project, two implementations of that interface, and the AdaptableAuction client and server projects.



3. Configure the two implementation projects, AAImplementations and AAImp2, as libraries. To do this, open the project workshop for each project. Select File | Configure as | Library... Then select File | Make Distribution... and perform a full make in the current environment. Once they have been distributed as libraries, AAImplementations and AAImp2 can be deleted from your workspace.
4. Run AdaptableAuction. Click the Calculate Tax for Sale Items button. This will write information to the log. The data used to load the library and find the class is written to the log. The log also shows information on each of three sale items, including the tax amount. In each of the three sale items, the tax is seven percent of the value of the bid amount. You can click the Calculate Tax for Sale Items button several times, to confirm that the same output is written to the log each time.
5. Do not exit the application. Copy the file dynload2.dat to FORTE\_ROOT/install/examples/frame/dynload.dat. Click the Calculate Tax for Sale Items button again. Note that a different library and class have been loaded this time. Also note, the second bidder's tax amount is now 0. The second implementation checked the bidder type. Since the second bidder was a nonprofit organization, he was not charged any taxes.

## AppletBanking

**Description** This application starts two applets that perform its functions: Banking and BankRecords.

**Pex Files** In the FORTE\_ROOT/install/examples/frame directory: apltbank.pex, banksvc.pex, banking.pex, bankrec.pex

**Mode** Distributed.

**Special Requirements** Banking and BankRecords must be installed in your environment as applets. The apltbank.fsc file contains the Fscript commands to import, make distributions, and autoinstall Banking and BankRecords as applets.

The AppletSupport library is a supplier to the AppletBanking project. If this library is not already in your repository, you need to import the apltsupp.pex file from the FORTE\_ROOT/userapp/appletsu/cl0 directory before you can import the apltbank.pex file.

You can run the AppletBanking example either as a test run from within the iPlanet UDS Workshops, or you can deploy the application.

► **To use AppletBanking**

1. Start the AppletBanking application.
2. Click a button to perform a function:

Button	Description
Manage Accounts	Starts the Banking application in a separate window.
View Account Data	Starts the BankRecords application in a separate window.
Quit	Stops the AppletBanking, and also stops Banking and BankRecords, if they have been started by AppletBanking.

When a button is clicked, the application invokes the RunApplet method on the LaunchService service object to start the appropriate applet.

For more information about how this application works, compare this example to the banking applet you create in the tutorial, which is located in *Getting Started With iPlanet UDS*.

## Auction

**Description** Auction illustrates prominent features and capabilities of an iPlanet UDS distributed application: GUI independence, distributed processing, event handling, multitasking, and image handling. The application allows a number of bidders located at their respective computers to bid on a set of paintings being offered by an auctioneer located at some other computer. The Art Auction application provides a list of paintings available for bidding and notifies interested bidders when a price changes.

**Pex Files** frame/utility.pex, tool/imageprj.pex, tool/aucserv.pex, tool/auction.pex.

**Mode** Standalone or Distributed.

**Special Requirements** The image files used by this application must be located in FORTE\_ROOT/install/examples/images.

➤ **To use Auction**

1. Start up the auction by clicking the Be Auctioneer option in the radio list, then clicking the Start Auction button.
2. Assume the role of a bidder by clicking the Be Bidder option in the radio list. You should click on a painting in the array, then click the View Painting button.

From the painting window, you can double-click on the image to see it enlarged. You can also click the Bid button to set a bid.

3. Another bidder can view available paintings being offered and then join the bidding process.

Both bidders become involved in bidding on the same painting. In the standalone use of this application, you can simulate a second bidder on the same screen by opening a second bidding window.

## AutoTester

**Description** AutoTester enables you to create test suites of iPlanet UDS GUI applications. It shows how to use the Capture and Playback class in the Display Library. You may find it covers all your automated testing needs. You may want to modify it for special testing purposes, or just use it as a reference when creating your own test utility.

**Pex Files** display/autotest.pex.

**Mode** Standalone or Distributed.

**Special Requirements** None.

➤ **To use AutoTester**

1. Refer to [Chapter 5, “Testing the User Interface”](#) for a complete description of how to run AutoTester on the PencilPlay sample program.

## Banking1-2

**Description** Banking1-2 example illustrates iPlanet UDS's interoperability features. It creates an old server, a new server, an old client and a new client, and allows you to run any combination of client and server.

**Pex Files** frame/banking1.pex, frame/banking2.pex, frame/banksvc1.pex, frame/banksvc2.pex, interop.fsc.

**Mode** Distributed only.

**Special Requirements** None.

### ► To use Banking1-2

1. Run Fscript and include interop.fsc.

```
fscript -i interop.fsc
```

2. Start a new server, and run both old and new clients with it.

```
-- Use the following commands to start servers and clients:
-- To start a new server:
ftexec -fi bt:$FORTE_ROOT/userapp/bankserv/cl0/bankse1 -ftsvr 0
-- To start an old server:
ftexec -fi bt:$FORTE_ROOT/userapp/bankser0/cl0/bankse1 -ftsvr 0
-- To start a new client:
ftexec -fi bt:$FORTE_ROOT/userapp/banking/cl0/bankin0
-- To start an old client:
ftexec -fi bt:$FORTE_ROOT/userapp/banking0/cl0/bankin0
```

3. Start an old server, and run both old and new clients with it.

## HTTPSupport

**Description** These examples illustrate iPlanet UDS client and server applications that implement the HTTP/HTTPS protocols. The examples do not represent a robust implementation of HTTP/HTTPS, but serve as illustrations on how to configure and run server and client applications. Typically, a server partition is configured for either HTTP or HTTPS. It is recommended you restart a server partition when switching between the server and secure server examples.

**Pex Files** These examples have the following .pex files

Pex File	Location	Description
HTTPClientExample.pex	extsys/http/client/	HTTP client
HTTPSClientExample.pex	extsys/http/client/	HTTP secure client
HTTPServerExample.pex	extsys/http/server/	HTTP server
HTTPSServerExample.pex	extsys/http/server/	HTTP secure server

**Mode** Standalone or Distributed.

**Special Requirements** None

► **To use HTTP examples**

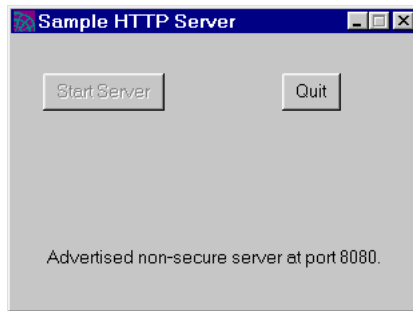
1. Import the .pex file for a server application into your workspace, and run the application.

Before running the server, make sure you restart the server partition if you previously ran an HTTP server example in that partition.

When you run a server example, the following window opens:



2. Start the server application by clicking the Start Server button.

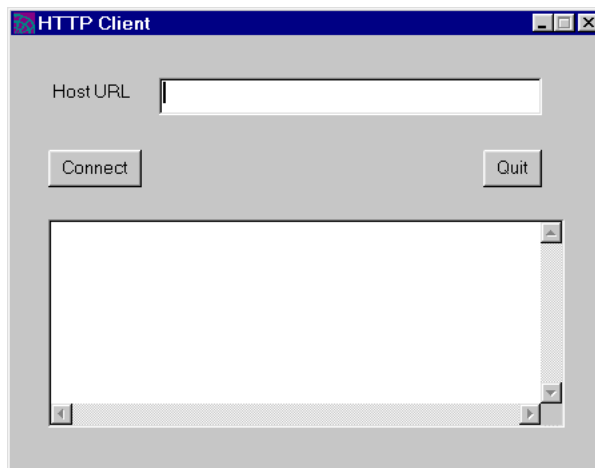


The server application dialog indicates the port the server is listening on.

3. Import the .pex file for the corresponding client application, which is typically on a different node or even a different environment, and run the client.

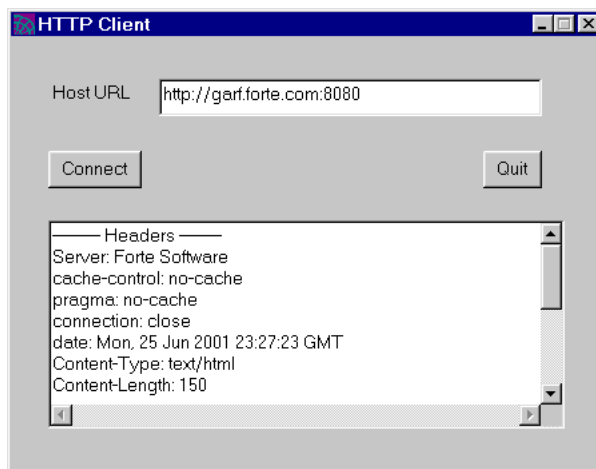
For example, if you ran the HTTPServerExample, then run the HTTPClient example.

When you run a client example, the following window opens:



4. Type in the fully qualified domain name of the HTTP Server, including the port number advertised in the HTTPServer window, and click Connect.

The HTTP Client displays HTTP headers and text from the HTTP server.




---

**NOTE** You can also test the iPlanet UDS HTTP servers and clients against third party web client and servers. In most cases, the third party applications can communicate with these basic examples.

---

## InheritedWindow

**Description** InheritedWindow shows how to create subclasses of your own UserWindow classes. In this example, the parent window has decorative widgets at the top, and three buttons at the bottom. It is a generic data entry window. Two windows are subclassed from this parent window. One is for entering information about art, the other about artists. They both use their own event handlers to validate data, and call the parent window's event handler to perform exit processing.

**Pex Files** display/inherwin.pex.

**Mode** Standalone or Distributed.

**Special Requirements** None.

➤ **To use InheritedWindow**

1. The first window lets you call up the Art Data Entry window or the Artist Data Entry window. They are spawned as tasks, so you can have them both open at once. They both have field and cross-field validation.
2. In the Artist window, try entering various countries. When you click OK, you will be warned that only certain names are compatible with that country. In the Art window, try entering various types of art.
3. Try entering 'Performance' as the type and '1910' as the year, then click the OK button.

In both windows, the Reset button will restore the original data, and the Cancel button will let you exit even if the data is invalid. The OK button exits if the data is valid, or keeps you in the grid if it is not.

## InternatBank

**Description** InternatBank shows how to use several of the iPlanet UDS international features. It can initially be brought up in several languages, and the user can choose to switch languages at runtime. It uses message catalogs to translate windows and error and informational messages.

**Pex Files** internat/internat.pex.

**Mode** Standalone or Distributed.

**Special Requirements** Copy files from FORTE\_ROOT/install/examples/internat subdirectories to corresponding directories under FORTE\_ROOT/workmsg.

➤ **To use InternatBank**

1. If you are running this application on a machine other than a PC or Macintosh, be sure your FORTE\_LOCALE environment variable is set correctly.
2. Below the examples directory, you will find a subdirectory called internat. Below it are the directories for each of the languages supported by the InternatBank example: en\_us, fr\_fr, and de\_de. In each of these directories is a file called internat.cat. This is the compiled message catalog for each language. You will also see files named int\_mac.msg and internat.msg. These are the message files in text format, composed on the MacIntosh or any other platform. Examine the text file that is compatible with the machine you are on. The compiled message file, internat.cat, is portable across all platforms.



3. Before running InternatBank, you will need to copy the `internat.cat` files from the language subdirectories under `internat` to the appropriate subdirectories under `FORTE_ROOT/workmsg`.

For this application, you should create three subdirectories under `workmsg`: `en_us`, `fr_fr`, `de_de`.

Copy the appropriate `internat.cat` files from the example program language subdirectories to the `workmsg` subdirectories.

4. Choose the language you want the application to start in. At the first window, select the language you wish to use.

You will see the screen change to that language immediately.

5. When you enter a numeric id, and click OK, the next screen will appear in the chosen language. Numeric, date time, and money fields should display values with appropriate formatting. You can move the Account Information window aside, since it is started as a separate task, and change the language on the first window. When you click OK, the Account Information window will come up in the newly requested language.
6. To see the application start in French, exit iPlanet UDS, set the language/territory component of your `FORTE_LOCALE` environment variable to `fr_fr`. If you are using a Macintosh or PC, use the Control Panel to change the `FORTE_LOCALE`. Restart iPlanet UDS and run InternatBank. The first screen will be in French. You can still change to other languages at runtime.

## NestedWindow

**Description** NestedWindow illustrates multiple tasks, nested windows, event handlers, and input validation. A launch window allows you to start two windows. Each of these windows nests another window. Field and cross-field input validation techniques are demonstrated.

**Pex Files** `display/nestwin.pex`.

**Mode** Standalone or Distributed.

**Special Requirements** None.

► **To use NestedWindow**

1. Click on the Purchase Art button. When the window comes up, move it aside.
2. On the first window, click on the Sell Art button. Notice that the same window is nested in both. They should come up with different initial values in the Type of Art field. This data was passed to the nested window's event handler as a parameter.
3. Experiment with entering different values in the Type of Art field and the Year field. There is input validation on the Type of Art field. There is cross-field validation between the Type of Art field and the Year field. Enter 'Performance' in the Type of Art field, and '1900' in the Year field, then click the OK button.

## NomadicOrderClient

**Description** This application starts a standalone client. In standalone mode, you can enter orders and search for entered orders on a local database based on the customer ID. You can also search the server database for customer orders and reconcile the local database with the server database. When you search for entered orders in the server database, the client partition connects to the name server and the service object. The client stays connected until you close the Find Order window. When you reconcile with the server (upload new orders and download a new copy of the server database to the client), the client connects to the service object, then disconnects when it is done.

**Pex Files** FORTE\_ROOT/install/examples/frame/nomad.pex

**Mode** Distributed

**Special Requirements** For more information about testing nomadic applications, see ["Testing Nomadic Client Applications" on page 451](#). You need to deploy this application to actually connect to and disconnect from the environment.

► **To use NomadicOrderClient**

1. Start the application using the **-fnomad** flag, for example:

```
ftexec -fi c:\forte\userapp\nomadico\cl0\nomadi0 -fnomad
```

2. Enter one or more orders, remembering the customer IDs.

3. Perform the following actions, which cause the client to connect to the environment, then disconnect:
  - a. Reconcile the client with the server.
  - b. Search for orders on the server.

## PrintSample

**Description** PrintSample shows how to use the printing classes. There are six subsections to the application; each one shows how you might solve different printing problems.

The first option, SimplePrint, uses no TOOL code. It relies on the **Print** and **PrintSetup** menu commands to print the current UserWindow.

The second option, SimpleClone, clones the current UserWindow and makes some changes to the cloned window's data. It also shows how to force changes to the DefaultPrintOptions object.

The third option, Expand&Tile, prints the current UserWindow, expanding all the fields so that their hidden data as well as their displayed data will be printed. When all the fields are expanded, page tiling will occur.

The fourth option, Report, uses a template window with header and footer information to print the data from the current window. The data from a large array is printed on a multi-page report. This part of the example shows how to create page and line breaks, while using DrawMultiLineText.

The fifth option, EmptyPage, uses the EmptyPage class as the WorkingPage.

The sixth option, ExpandText, prints a TextData object that contains long lines. DrawMultiLineText is used with different height and width policies on the PrintDocument.

**Pex Files** display/printsam.pex.

**Mode** Standalone or Distributed.

**Special Requirements** Printer connection. The files artist.dat and painting.dat must be located in FORTE\_ROOT/install/examples/database. The file hr.dat must be located in FORTE\_ROOT/install/examples/display.

### ► To use PrintSample

1. From the main window, choose the buttons that sound of interest. Each subsection allows you to print and to set up print options. Some have more refined options.
2. Report lets you print the array without the report template, using standard array field expansion and vertical tiling, for comparison. It also offers two styles for the report: one where it is measured in mils, another where it is measured in columns.
3. The ExpandText option lets you see the use of DrawMultiLineText with and without tiling and with various line spacing options. You can choose between natural and explicit height and width policies to observe horizontal and vertical tiling, as well as multi-page printing without tiling. You can also choose various line spacing options.

## TabFolders

**Description** TabFolders illustrates how to use the TabFolder class, and how to use Popup Menus. The example allows you to open two windows. One window illustrates basic TabFolder and Popup functionality. The TabFolder widget was constructed in the Window Workshop. The second window illustrates how to create a TabFolder dynamically and how to show Popup Menus dynamically.

**Pex Files** display/tfolder.pex.

**Mode** Standalone or Distributed.

**Special Requirements** None.

### ► To use TabFolders

1. The first window presents two pushbutton fields: Basic TabFolder and Dynamic TabFolder. Click Basic TabFolder.
2. A TabFolder with three tab pages appears on the Basic TabFolder window. The second tab page is the top page. Bring other tab pages to the front by clicking on them.
3. On the Bring To Front (Popup Demo) pushbutton, use the key combination required by your client operating system to activate the Popup Menu. You will see a submenu with three menu items: Tab One, Tab Two, and Tab Three. Click any of these menu items, and that tab page will become the top tab page in the tabfolder.

4. Back in the first window, click the Dynamic TabFolder pushbutton. Bring each tab page to the front by clicking on it. Turn the Show Tree Tab Page toggle off. The Tree Tab Page will disappear from the TabFolder. Turn the Show Tree Tab Page toggle back on. The Tree Tab Page will reappear in the TabFolder.
5. The Popup Demo 1 and Popup Demo 2 pushbuttons each have Popup submenus associated with them. The Bring To Front (Dynamic Popup Demo) pushbutton will present a different popup menu depending on whether the Tree Tab Page is visible or not visible. Try it under both conditions.

## TimeItV1-4

**Description** The files `timeitv1.pex`, `timeitv2.pex`, `timeitv3.pex`, and `timeitv4.pex` are different versions of the TimeIt application, used to illustrate run-time compatibility issues. `timeitv2.pex` is compatible with `timeitv1.pex`, while `timeitv3.pex` and `timeitv4.pex` are not.

**Pex Files** `frame/timeitv1.pex`, `frame/timeitv2.pex`, `frame/timeitv3.pex`, `frame/timeitv4.pex`.

**Mode** Distributed only.

**Special Requirements** None.

### ► To use TimeItV1-4

1. You will build distributions from the different versions of TimeIt supplied, and install partitions from different distributions on the client and the server. For information on runtime compatibility, see [“Class Runtime Properties” on page 325](#).

### ► To use TimeItV1

1. Import `timeitv1.pex` into your workspace.
2. Partition it so that the client partition (`Timeit_cl0_Client_<node>`) id is on the client machine and the server partition (`Timeit_cl0_Part1_<node>`) is on the server machine.
3. Make a distribution.
4. Install the distribution on both the client and server machine.

5. Run the server partition on the server machine and the client partition on the client machine.

You'll see the usual TimeIt example application. Pressing the Start button makes the time display every second and beep at the minute. Now exit the client partition and kill the server partition.

► **To use TimeItV2**

1. Import `timeitv2.pex` into the same workspace.
2. This time partition it with both partitions on the server machine and install it only on the server machine.
3. Run the server and client partitions on the server machine. There is one new feature: you can optionally beep at the quarter minute as well as the minute.

The following changes were made to TimeItV1:

- The class `Clock` has a new attribute `BeepOnQuarterMinute`. It has a higher serial number than any previously existing attributes had.
- `Clock` has a new event, `DoBeep`.
- `Clock` has a new overloading of the method `StartClock`.
- `ClockWindow` has changed. It includes a new attribute `ToBeep` generated by the new button.

None of the existing IDs have changed, and all the rules for adding things have been obeyed. The partitions from TimeItV1 and TimeItV2 should be compatible. To test this, run the client partition from the client machine, which is still the version from TimeItV1. Note that it works correctly. It cannot access the new feature of beeping on the quarter minute, but the existing features work correctly.

► **To use TimeItV3**

1. Import `timeitv3.pex` into the same workspace.
2. Again, partition it so that both partitions run on the server machine, and install it only on the server.
3. There is one important difference between TimeItV1 and TimeItV3 that will cause an incompatibility: instead of adding a new overloading of `StartClock`, TimeItV3 replaces the old `StartClock`, which had no parameters, with a new one, which has a boolean input parameter. Since the overloading of the method which the older client uses no longer exists, TimeItV3 is incompatible with TimeItV1.

4. You can see this by running the new server partition on the server machine and the old client partition on the client machine. When you click the Start button on the client, which calls the method, you get the error:

```
SYSTEM ERROR: No actual parameter for argument 1 of StartClock
```

This happens because the client partition doesn't pass a parameter, but the server expects one.

#### ► To use TimeItV4

1. Import `timeitv4.pex` into the same workspace.
2. Again, partition it so that both partitions run on the server machine, and install it only on the server.
3. There is one important difference between `TimeItV1` and `TimeItV4` that will cause an incompatibility: `TimeItV4` adds a parameter to the event **SecondElapsed** which is used to communicate between the client and the server. This makes the new server partition incompatible with the old client partition.

To see this, run the new server partition and the old client partition. Click the Start button. The first time **SecondElapsed** is sent to the client partition, the result is:

```
SYSTEM ERROR: After processing event <xxx> the stack is
incorrectly set.
```

iPlanet UDS's interpreter's stack is incorrect because the event was passed an unexpected number of parameters.

## TreeList

**Description** The `TreeList` example shows how to coordinate the display of data in `TreeView` and `ListView` fields. The `TreeView` field displays a hierarchy of biological classifications: orders, families, and genera. The `ListView` field displays detailed information on species within a genus.

**Pex Files** `display/tv/v.pex`.

**Mode** Standalone or Distributed.

**Special Requirements** none.

➤ **To use TreeListExample**

1. Click on the controls in the TreeView field to expand and collapse the outline. Not all the nodes have children, but by opening them all you will see order, family, and genus names for some Costa Rican birds.
2. When you click a node at the genus level, you will see the species in that genus displayed in the ListView field.



## SYMBOLS

- .a file 314
- .ace file 274, 314
- .adf file 274, 314
- .bom file 314
- .btd file 274, 314
- .btx file 274, 314
- .cc file 314
- .cdf file 314
- .dll file 314
- .exe file 274, 314
- .fso file 314
- .lgf file 314, 315
- .out file 177
- .pex file 314
- .pgf file 274, 279
- .so file 314

## A

- About command 162
- AboutMenuActivate event 162
- Active partition
  - log file names, changing 212
- Active window 59
- Add Project command 310
- Advertise method
  - HTTPHelper class 567
- AfterTabSelect event 83
- AgentAccess sample application 632
- Alignment
  - column alignment 141
  - labels in a grid field 138
  - row alignment 141
  - with grid fields 140
- Alignment property 95
- Allowed property 328
- Anchored object 330
  - named 332
  - non-distributed 332
  - unnamed 332
- Applet
  - application distribution and 271
  - configuring an application as 442
  - creating 256
  - defined 226
  - deploying applications with 444
  - designing applications with 437
  - testing 443
  - troubleshooting clients 444
  - visibility of 444
- AppletBanking application 633
- AppletSupport library
  - about 435
  - advantages of 436
  - restrictions 437
  - setting up 436

## Application

- compatibility level of shared libraries 480
- configuring with applets 442
- designing with applets 437
- installing 297
- installing on client nodes 299
- installing on server nodes 298
- installing with reference partitions 303
- partitions, upgrading single 323
- reference partitions, upgrading 321
- removing 319
- testing with applets 443
- upgrading 320
- upgrading distributed 456

## Application distribution

- about 270
- components of 272
- creating 315
- deploying 286
- documentation 285
- file naming conventions 275
- generated files 284
- installing 296, 297
- installing additional files with 284
- installing on client nodes 299
- installing on server nodes 298
- loading 288
- making 284
- packaging 284
- partition properties, changing 295
- partitioning configurations, modifying 292
- partitions, reassigning 293
- transferring 287
- uninstalling 319

## Application session

- HTTP session 548

## AppTitle attribute 162

## array

- getting information about 503

## Array class

- and IsAnchored attribute 335

## Assigned partition

- deleting 265
- examining 246

## Asynchronous processing

- HTTPSupport library 556

## Attr class 600

## Attr.setValue() method 598

## Attribute

- adding 459, 477
- deleting 459
- remote access 487
- runtime properties for 329

## Attribute Name property 94

## AttributeList interface 611

## Attributes interface 611

## Auction sample application 634

## aud message type 186

## Auto-install

- Make Distribution command and 278
- on a Macintosh or PC 278
- using with installed applications 278

## Automatic partitioning 227, 269

## Auto-starting service objects 263

## AutoTester 167–180

- about 167
- analyzing results 178
- automating regression tests 179
- capturing input 170
- dumping state information 172
- DumpState method 173
- playing back 175
- setting up 169

## AutoTester sample application 635, 636

**B**

## Banking1-2 application 466472–475

## byte offset

- in DOMString datatype 601
- in text nodes 599

**C**

## Captured input 167

- playing back 175

- case statement
  - in converter 486
- Casting, interfaces 368
- CDATASection class 600
- cfg message type 186
- cfg:em:2 302
- CharacterData class 600
- CharacterData.insertData() method 599
- CharacterData.replaceData() method 598, 599
- CharacterData.setData() method 598, 599
- CharacterData.substringData() method 599
- Characters() method 608
- Child event 65
- Class
  - adding 459
  - deleting 459
  - distributed property 326
  - listed by partition 481
  - loading dynamically 371
  - monitored property 326
  - shared property 326
  - transactional property 326
  - version 469
  - version runtime property 461
- class
  - getting information about 501
- class browser 513
- Class runtime properties
  - definition 325
  - for cloned copies 330
  - for nested object references 329
  - performance 327
  - setting 328
  - setting default 328
- Class version
  - definition 460
- Class version upgrade 454
  - changes allowed 459, 467
  - described 465
  - installing new partitions 493
  - making distribution 493
  - steps 481
- Classversion key word 486
- Client configuration
  - defined 226
- Client node
  - installing applications on 299
- Client partition 229
  - Applet property 256
  - Compiled property 265
  - Generate C++ API property 266
  - icons, automatically generated 301
  - icons, defining 302
  - testing as 163
- Clone method 330
- Cloning
  - shared objects 343
  - transactional objects 356
- Code generation
  - fcompile command 281
  - for libraries 315
  - for partitions 279
- Column alignment 141
- Column Name property 95
- Column partnership
  - and grid fields 145
  - commands 145
- Column Title property 95
- Column Title Set Number property 95
- Column weight 141
- Command, prefabricated *See* Prefabricated command
- Compatibility level 454
  - definition 460
  - library 461
- Compatibility level upgrade 455
  - changes allowed 459
  - described 463
  - rolling 464
  - steps 478
- Compilation Properties for Node dialog 311
- Compiled partition
  - about 270
  - making a distribution 279
- Compiled property 295
  - client partition 265
  - server partition 267
- Compiling
  - libraries 311, 315
  - partitions 279

- Configuration
    - about 225–268
    - applet and 226
    - changing for an installed application 278
    - changing for installed applications 321
    - client 226
    - debugging 248
    - examining 269
    - modifying 269
    - properties 268
    - re-partitioning 269
    - server 226
    - testing 248
    - viewing properties 268
  - Configure as Library command 234, 304, 461
  - Connected environments, reference partition in 254
  - Constant, in interface 370
  - ContentHandler interface 612
  - ContentHandler.skippedEntity() method 611
  - Context-sensitive help 149
  - Converters 467–472
    - deleting 491
    - event 471
    - inherited 486
    - name 485
    - signature 485
    - testing 493
    - viewing 483
    - when required 465, 467
    - writing 485–491
  - Cookie
    - HTTPSupport class 552
  - Cookies
    - defined 549
  - Copy, object 331
  - Cross-environment failover 416
- ## D
- DataValue class
    - and IsAnchored attribute 335
    - and isTransactional attribute 344
  - DBResourceMgr class 394
  - DBResourceMgr service object 396
  - DBSession class 394
  - DBSession service object 396
    - database name 397
    - user name 397
    - user password 397
  - DCE (external type) 262
  - Deadlock
    - avoiding 340
    - common mutex 339
    - distributed mutex 340
    - lock promotion 352
    - transactional 351
  - Debugger, using from Partition workshop 248
  - Default help file 149
  - Default Node property (configuration) 268
  - DefaultHandler class 608, 612
  - DefaultHelpFile attribute 149
  - Deployment environment 234
  - Development environment
    - description 234
  - Dialog box 63
    - appearance of 131
    - file selection 63
    - message 63
    - print 63
    - print setup 63
    - question 63
    - related Window class methods 131
    - using 63
  - Dialog duration 402
    - message duration 405
    - session duration 409
    - transaction duration 407
  - Disabled property 267, 295
  - Disallowed property 328
  - Display method 58
    - event loop for 64
    - invoking on window 60
  - DisplayNode class
    - outline field and 88
    - using for list view field 97, 104, 113
    - using for tree view field 116
  - Distributed application
    - updating 460

- Distributed class
    - updating 460
  - Distributed object 330
    - access to attributes 333
    - IsAnchored attribute 330
    - methods invoked on 332
    - reference 330, 469, 476
    - shared 342
  - Distributed property 326, 330
  - Distribution
    - full 477
    - partial 477
    - source repository 478
    - upgrading a 477
  - Distribution directory 272
  - Distribution, application
    - about 270
    - applets and 271
    - directory 272
    - icon file 272
    - making 269
  - Distribution, library
    - about 312
    - directory 312
    - making 312
  - Document class 600
  - Document.createAttribute() method 597
  - Document.createAttributeNS() method 597
  - Document.createCDATASection() method 597
  - Document.createComment() method 597
  - Document.createDocumentFragment() method 597
  - Document.createElement() method 597
  - Document.createElementNS() method 597
  - Document.createEntityReference() method 597
  - Document.createProcessingInstruction()
    - method 597
  - Document.createTextNode() method 597
  - DocumentFragment class 600
  - DocumentHandler interface 612
  - DocumentType class 600
  - DOM
    - defined 595
  - DOM exception handling 601
  - DOM nodes
    - text nodes 599
    - tree nodes 595
  - DOM trees 595
    - methods for creating objects 597
  - DOMException class 600
  - DOMImplementation class 600
  - DOMImplementation.createDocument()
    - method 597
  - DOMImplementation.createDocumentType()
    - method 597
  - DOMString datatype 601
  - DTDHandler interface 612
  - DumpState method 173
  - DumpWidget method 172
  - Dynamic class loading 371
- ## E
- Element class 600
  - Encina (external type) 262
  - EndDocument() method 608
  - EndElement() Method 608
  - Entity
    - HTTPSupport class 552
  - Entity class 600
  - EntityReference class 600
  - EntityResolver interface 612
  - Environment search path 432–434
    - on a service object 262
    - reference partition and 254
    - service object definition 426
    - syntax 263
  - Environment visibility 400
  - Environment, definition 234
  - err message type 186
  - ErrorHandler interface 612
  - Event
    - close window event chain 68
    - completion 472
    - converter 471
    - deleting 459

- Event (*continued*)
    - exception 472
    - field selection event chain 67
    - in interface 370
    - input focus chain 65
    - replacing 459
    - shared object and 337
    - shared objects and 337
    - transactional objects and 346
  - Event converter
    - referencing attributes 489
    - writing 489
  - Event handler
    - adding 459
    - deleting 459
    - in interface 370
    - inherited windows and 50
  - Event statement 64
  - Excluded Node property (configuration) 268
  - Export name 262
  - Export Name property 262
  - ExportMode 533
  - ExportPlan command (Fscript) 254, 291
  - External type 262
- F**
- Failover 413–420
    - cross environment 416
    - definition 413
    - during upgrade 457
    - example 463
    - in class version upgrade 466
    - local 414
    - partition 261
    - replicates in class version upgrade 495
  - fcompile command
    - environment variables, used with 279
    - for libraries 316
    - syntax 281–283
  - File export formats
    - compatibility format 533
    - multi-file format 534
    - overview 533
  - File menu command, prefabricated 133
  - File names
    - in log flag settings 201
    - O/S conventions 202
    - stdout definitions 210
    - stdout, stderr, and stdin 201
  - File selection dialog box 63
  - FileOpenDialog method 131
  - FileSaveDialog method 132
  - FilterImpl.SetContentHandler() method 609
  - FilterImpl.SetParent() method 609
  - FindClass method 378
  - FindLibrary method 377
  - findsubagent command 494
  - First/Last property 95
  - fl flag 185, 198
  - Float-over help 152
    - for palette lists 154
  - Flush method 196, 217
  - FlushLogFiles command 195
  - fns flag 447
  - Font
    - portable 129
    - using in windows 129
  - FORTE\_AUTOTESTER\_DELAY 177, 179
  - FORTE\_AUTOTESTER\_ROOT 174
  - FORTE\_LOGGER\_SETUP 197
    - description 185
    - example 218
    - overriding with -fl flag 201
  - FORTE\_NS\_ADDRESS 447
  - FORTE\_ROOT 482
  - fs flag 447
    - nomadic client and 450
  - ftcmd run command 302
  - Ftcmd utility 445
  - ftexec command 302

**G**

- Generate C++ API property 266
- GenericRepository
  - classes 529
  - SCMServer 529
- GenericRepository Library 527
- Grid field
  - column alignment 140
  - column partnerships 145
  - column weight 141
  - moving 145
  - nesting 137
  - row alignment 140
  - row partnerships 145
  - row weight 141
- Group Into TabFolder command 74
- Group number (logging filter) 188

**H**

- HandlerBase class 612
- Has Column Titles property 95
- Has Controls property 95
- Has Horizontal Scrollbar property 95
- Has Row Highlights property 95
- Has Vertical Scrollbar property 95
- Header Style property 79
- HeightInPixels attribute 131
- Help 147–159
  - About command 162
  - AboutMenuActivate event 162
  - AppTitle attribute 162
  - commands 160
  - context-sensitive 149
  - context-sensitive help 149
  - Default help file 149
  - default help file 149
  - DefaultHelpFile attribute 149
  - float-over help 152
  - for menu widgets 159
  - for palette lists 154, 158
  - Help Text command 148, 151, 153
  - HelpRequest event 148
  - IsFloatOverEnabled attribute 152
  - prefabricated help commands 160
  - status-line help 155, 159
  - StatusText attribute 148
  - WinHelp method 148
- Help Text command 148, 151, 153
- HelpRequest event 148
- High availability applications 456
- HorzPixelsPerInch attribute 131
- HTTP protocol
  - application sessions 549
  - entity 547
  - headers 546
  - HTTPSupport model 544
  - message body 547
  - messages 546
  - messaging model 543
  - network sessions 550
  - secure sessions 550
  - sessions 548
- HTTPHelper class
  - Advertise method 567
- HTTPSupport library
  - about 543
  - asynchronous processing 556
  - classes 552
  - configuring secure sessions 575
  - creating client applications 553
  - creating server applications 559
  - dispatching requests 556
  - Entity class 565
  - HTTPServerManager interface 574
  - interfaces 552
  - processing cookies 563
  - request messages 553
  - responding to requests 562
  - sample application 636
  - synchronous processing 556
- HTTPSupport library examples
  - advertising a server 561
  - building a client request 553
  - configuring a client 568
  - configuring a server 571
  - creating a session 558
  - processing the body of a message 564

## HTTPSupport library examples (*continued*)

- registering a request 566
- registering an entity 566
- response message from server 559
- reusing a session 558

## I

### Icon

- creating for client partitions 303
- file in application distribution 272
- generated for compiled client partitions 302
- generated for standard client partitions 301

### Image resolution 130

### Implementation library

- about 384
- creating 386
- registering 388

### Inactive window 59

### Inherited window

- definition 48
- overview 49

### InheritedWindow sample application 639

### Init method 328, 329

### Input capture

- capturing 170
- output file 168
- playing back 175

### Input focus 66

### InputDriver method (input capture class) 168

### InputSource class 613

### Installing

- additional files with a distribution 284
- application distributions 296
- library distributions 318
- reinstalling using auto-install 278
- using auto-install on a Macintosh or PC 278
- with Make Distribution 278

### Interface

- about 359
- casting 368
- changing 459

### constants in 370

- creating 362
- defining 373
- elements of 368
- event handler in 370
- event in 370
- FindClass method with 378
- FindLibrary method with 377
- implementing 364, 384
- loading class 376
- loading class implementation 371
- method in 369
- multiple inheritance with 389
- testing 389
- using as a type 367
- virtual attribute in 368

### interface

- getting information about 501

### Interface library

- about 373
- creating 380
- deploying 387
- importing 383

### Interface Properties dialog 363

### Interface Workshop 359

### InternatBank sample application 640

### Interoperability 462

### Interoperable upgrade 454

- changes allowed 459, 476
- described 462
- full or partial distribution 477
- steps 475

### INVALID\_ACCESS\_ERR 601

### INVALID\_MODIFICATION\_ERR 601

### INVALID\_STATE\_ERR 601

### IsAnchored attribute

- and Datavalue classes 335
- definition 326
- distributed object and 330
- IsShared attribute and 331, 342
- setting 329

### IsDefault property 328

### IsFloatOverEnabled attribute 152



IsShared attribute  
 definition 326  
 IsAnchored attribute and 331, 342  
 setting 329  
 shared objects and 335  
 with mutex 337

IsTransactional attribute 329  
 definition 327  
 Transactional class property and 344

## L

Launcher application, testing 443

LaunchMgr class  
 about 438  
 methods 439

LaunchService service object 439  
 about 438  
 methods 439

Layout Policy property 79

LDAP library  
 about 615  
 closing a session 628  
 establishing sessions 617  
 LDAP filters 621  
 searching a directory 619  
 updating a directory 623  
 using 617

LDAP library examples  
 adding an entry to a directory 624  
 authenticating a session 618  
 building filters 621  
 closing a session 628  
 connecting to a server 618  
 deleting a directory entry 627  
 modifying a directory entry 626  
 searching a directory 620

Level number (logging filter) 188

Library  
 about 225–234  
 changing compatibility level 479  
 compatibility level 461  
 compiling 311, 315  
 interface 373  
 making distribution for 312  
 name for 309  
 removing 319  
 upgrading installed 322  
 upgrading references to upgraded 323

Library application 233

Library configuration  
 adding projects 309  
 creating 304  
 defined 308  
 examining 306  
 modifying 308  
 removing libraries from nodes 310

Library distribution  
 about 312  
 components of 312  
 installing additional files with 284  
 making 312  
 uninstalling 319  
 upgrading 322  
 UUID (universally unique identifier) 291

List view field 102–115  
 column properties 91, 107  
 column titles 91, 106  
 compared to outline field 102  
 creating in Window Workshop 93, 108  
 data for 88, 104  
 interacting with 88, 104  
 mapped type for 93, 108  
 portability 103  
 properties dialog for 94, 110  
 properties for 90, 105  
 providing data for 96, 112  
 row highlights 90, 106  
 scroll policy 90, 106  
 scrollbars for 90, 105  
 SetViewNodes method 115  
 sorting 103  
 styles 102

List view, on different platforms 128

Load balancing 420–426  
 in class version upgrade 495  
 partition 261  
 router partition 424

- load balancing
  - performance based 537
  - setting node performance 538
  - using performance-based 539
- Load Distribution command 290
- Local failover 414
- local name 594
- Locator interface 613
- LocatorImpl class 613
- Lock promotion 351
  - avoiding deadlock 354
  - blocking during 351
  - deadlock 352
- Locking
  - common mutex deadlock 339
  - distributed recursive deadlock 341
  - lock promotion 351
  - mutex 336
  - nesting transactions 355
  - shared object 336
  - start task statement 355
  - transactional deadlock 351
  - transactional locking 347
- Log files
  - flushing 195–196
  - test run, distributed 209
  - test run, local 209
- Log flag settings
  - changing with ModifyFlags method 199
  - command line syntax 201
  - defining stdout 199
  - definition of stdout 210
  - example 216
  - file name conventions 202
  - file names 201
  - order of precedence 200
  - special note for NT 200
  - syntax 201
- Log flags
  - definition of current settings 190
  - filters, *See* Logging filters
  - fl flag 185
  - performance example 219
  - setting in Repository Workshop 198206–207
  - setting with Econsole 198206–207
  - setting with Escript 199
  - setting with -fl flag 198
  - setting with FORTE\_LOGGER\_SETUP 197
  - setting with partition properties 198
  - setting with the Control Panel 197206–207
  - wildcards (command line) 202
- Log Flags property 197
- Log Flags tab page (Control Panel) 197206–207
- Logging
  - assigning group numbers 188
  - assigning level numbers 188
  - assigning message types 185
  - assigning service types 186
  - capturing output on a Macintosh 210
  - examining requirements 183
  - examples 214–221
  - flushing log files 195–196
  - iPlanet UDS tools for 184
  - location of output files 210–213
  - LogMgr methods 190–196
  - outputting state information 192
  - outputting text 192
  - procedures 181
  - setting flags 196–208
- Logging examples
  - Auction 214
  - LogTime method 221
  - performance 219
  - timer method 220
- Logging filters
  - changing 195
  - command line syntax 202
  - description 185–189
  - group number 188
  - level number 188
  - message type 185
  - service type 186
  - useful runtime filters 189
- Logging output 210
- Logical partition
  - about 229–232
  - creating 250
  - examining 244

LogMgr class 190–196  
 example 192, 194  
 Flush method 196  
 Flush method, example 217  
 ModifyFlags method 199  
 PutLine method 191, 193  
 Putline method, example 217  
 referencing LogMgr methods 191  
 Test method, example 217

## M

Make Distribution command 276–279  
 Auto-Compile option 277  
 for compiled partitions 279  
 for libraries 315  
 Full or Partial Make option 277  
 Install in Current Environment option 278  
 Local/Remote option 277

Managing Source Code 527

Mapped Type property 94

Menu bar, appearance of 132

Menu widget  
 status-line help 159

Message dialog box 63

Message dialog duration 405  
 error handling 406  
 events and 407  
 state information 406  
 transactions and 406

Message types (logging filters)  
 audit messages 186  
 configuration messages 186  
 error messages 186  
 performance messages 186  
 resource messages 186  
 security messages 186  
 trace messages 186

MessageDialog method 132

Method  
 adding 459  
 changing behavior 471  
 changing signature 485  
 converter 467  
 converter for new 470  
 deleting 459  
 in interface 369  
 referencing in converter 488  
 versions of 467

method  
 getting information about 508  
 invoking through reflection 509

Method converter  
 testing 493  
 writing 487

Method invocation statement 339

Minimum height 143

Minimum rows/columns 129

Minimum width 143

Modify Log Flags command 198206–207

ModifyFlags method 199

ModLogger command 199

ModLoggerRemote command 199

Monitored object 357

Monitored property (class) 326

MsgNumber property 95

Multiple inheritance 389

Multitasking  
 nonshared objects and 337  
 windows and 61

Mutex  
 and IsShared attribute 336  
 common mutex deadlock 339  
 deadlock, distributed 340  
 transactional lock and 347

## N

Named anchored object 332

NamedNodeMap class 600

NamedNodeMap.removeNamedItem() method 599

NamedNodeMap.setNamedItem() method 599

NAMESPACE\_ERR 601

## Natural

- size policy for character fields 129
- size policy for fields 136
- size policy for graphics 130

## Nested objects 329

- runtime properties for 329

## Nested window 61

- definition 48
- overview 51

## NestedWindow sample application 641

## Network session

- HTTP session 548

## New converter 469

## New ListView command 108

## New Logical Partition command 250

## New method converter 470

## New Reference Partition command 252

## New TabFolder command 73

## New TreeView command 120

## Node

- default 268
- examining 245, 307
- excluded 268
- properties 245

## Node class 600

## Node Outline command 245, 307

## Node.appendChild() method 599

## Node.cloneNode() method 599

## Node.getChildNodes() method 598

## Node.getFirstChild() method 598

## Node.getLastChild() method 598

## Node.getNextSibling() method 598

## Node.getParentNode() method 598

## Node.getPreviousSibling() method 598

## Node.insertBefore() method 599

## Node.normalize() method 599

## Node.removeChild() method 599

## Node.replaceChild() method 599

## Node.setNodeValue() method 598, 599

## NodeList class 600

## nodes

- setting performance of 538
- specifying performance information 539

## Nomadic client

- connecting to the environment 447
- defined 445
- disconnecting 448
- disconnecting from environment 449
- disconnecting from remote objects 448
- fns flag 447
- FORTE\_NS\_ADDRESS 447
- restrictions 452
- starting 450

## Nomadic client application

- about 445
- testing 451

## NomadicOrderClient application 642

## Non-distributed anchored object 332

## Non-replicated partition 230

## Nonshared object 337

## Normalize() method 599

## Notation class 600

## Number of Processors property 538

## O

## Object

- anchored 330
- Clone method 330
- distributed 330
- distributed reference to 476
- monitored 357
- nested 329
- shared 335
- transactional 343

## Object class

- runtime property attributes 326

## object inspector 510

## ObjectBroker (external type) 262

## ObjectLocationMgr class 332

## Obsolete converter 470

## Obsolete method converter 470

- example 468

- Outline field 88–101
    - Alignment property 95
    - Attribute Name property 94
    - Column Name property 95
    - Column Title property 95
    - Column Title Set Number property 95
    - DisplayNode class 88
    - First/Last property 95
    - Has Column Titles property 95
    - Has Controls property 95
    - Has Horizontal Scrollbar property 95
    - Has Row Highlights property 95
    - Has Vertical Scrollbar property 95
    - Mapped Type property 94
    - MsgNumber property 95
    - Root Displayed property 95
    - Scroll Policy property 95
    - Size Policy property (column) 95
    - State property (column) 95
      - when to use 88
    - Width property 95
- P**
- Page template
    - definition 48
    - overview 54
  - Palette list
    - and portability 131
    - float-over help 154
    - status-line help 158
  - Parent
    - size policy and row/column partnerships 145
    - size policy for character fields 129
    - size policy for fields 138, 142
    - size policy for grid fields 134, 136, 137
  - Parent attribute 61
  - Parser interface 613
  - ParserFactory class 613
  - Partial installation, completing 303
  - Partition
    - about 228
    - automatic startup 434
    - client 229
      - combining service objects on 257
      - compiled 270
      - Compiled property 265, 267
      - compiling 279
      - defined 228
      - deleting 265
      - Disabled property 267
      - distribution date 476
      - logical 229
      - moving 264
      - non-replicated 230
      - object anchored to 332
      - private 257
      - projects and 226
      - projects in 481
      - properties, changing 295
      - reference 230
      - replicated 229
      - Replication Count property 268
      - router 229
      - server 229
      - Server Arguments property 267
      - shared 258
      - standard 270
      - Thread Package property 267
      - transactions distributed across 356
      - upgrade order 494
      - upgrading applications 323
  - Partitioning
    - automatic 269
    - configurations, modifying 292
  - PDF files, viewing and searching 44
  - Performance Rating property 538
  - Picture button, and portability 131
  - Playing back input 175
  - Portable font 129
  - Prefabricated
    - help commands 160
    - submenu 132
  - Prefabricated command
    - Edit menu commands 133
    - Help commands 133
    - Print commands 133
  - prefix 594
  - prf message type 186
  - Print dialog box 63

Print setup dialog box 63  
 PrintDialog method 132  
 PrintSample sample application 643  
 Private partition 257  
 ProcessingInstruction class 600  
 ProcessingInstruction.setData() method 599  
 ProcessingInstruction.setNodeValue() method 599  
 Project
 

- export with IDs 478
- library name for 309
- listing in partitions 481

 Properties command
 

- Configuration properties 268
- Node properties 245, 307

 Property sheet, creating 132  
 PurgeEvents method 67  
 PutLine method 191, 193, 217

## Q

qualified name 594  
 Question dialog box 63  
 QuestionDialog method 132

## R

Read lock 349  
 Reference partition 426–431
 

- about 230–232, 426
- auto-start for 253
- changing compatibility level 479
- connected environment with 254
- creating 251
- defined 230
- examining 245
- installing applications with 303
- making 251
- supplier project for 251
- upgrading applications with 321

reflection
 

- accessing objects 500
- array information 503
- class hierarchy 499
- classes that enable 499
- method invocation 509
- methods 508
- restrictions on use of 498
- sample class browser 513
- sample object inspector 510
- setting attributes 505
- setting primitive types 506
- simple data types 504
- use of 497

 RegisterObject method 332  
 ReleaseConnection method 448  
 ReleaseNameService method 449  
 Repartition command 269  
 Replicated partition 229  
 Replicates
 

- for failover 413
- for load balancing 420
- in class version upgrade 495
- specifying number of 422
- using in class version upgrade 482, 494

 replication
 

- specifying on service object 539

 Replication Count property 268, 295, 542  
 res message type 186  
 Rolling upgrade 456–457
 

- class version upgrade steps 481
- using class versions 465
- using compatibility level 464
- when required 456

 Root Displayed property 95  
 Root node
 

- for tree view field 121

 Router partition 424
 

- description 229
- in class version upgrade 495
- multiple routers 259
- primary 412

 Router, custom 357  
 Row alignment 141

- Row partnership
  - and grid fields 145
  - commands 145
- Row weight 141
- Runit method 168
- Runtime properties (class) 325
  - attributes for 328
  - default 327
  - disabling for performance 327
  - for cloned copies 330
  - for nested object references 329
  - version 461

## S

- Sample applications
  - AgentAccess 632
  - Auction 634
  - AutoTester 635, 636
  - HTTPSupport 636
  - InheritedWindow 639
  - InternatBank 640
  - NestedWindow 641
  - NomadicOrderClient 642
  - PrintSample 643
  - scmserver 533
  - TabFolders 644
  - TimeIt 647
  - TimeItV1-4 645
- SAXException class 613
- SAXNotRecognizedException class 613
- SAXNotSupportedException class 613
- SAXParseException class 614
- SCM service
  - creating 532
  - installing the sample service 533
- SCM utility 531
- SCM\_EXPORT\_MODE\_COMPAT 533
- SCM\_EXPORT\_MODE\_MULTIPLE 533
- Scroll Policy property
  - list view field 91, 106
  - outline field 95
  - tree view field 119
- sec message type 186
- Secure session
  - HTTP session 548
- Secure Sockets Layer (SSL) protocol 550
- security 581
- Server arguments property 267, 295
- Server configuration 226
- Server node, installing applications on 298
- Server partition 229
- Server upgrade
  - high-availability 482
  - without interruption 463
- Service object 332
  - adding 459
  - auto-starting 263
  - changing compatibility level 480
  - combining on partitions 257
  - DBResourceMgr type 394, 396
  - DBSession type 394, 396
  - deleting 459
  - dialog duration 402
  - distributed property 330
  - environment search path 262, 432
  - environment visibility 400
  - examining in partition 245
  - export name for 262
  - Export Name property 262
  - external type for 262
  - External Type property 262
  - failover 413
  - load balancing 420
  - modifying in partition 247, 260
  - moving to a new partition 250
  - overview of 393
  - reference partition for 426
  - TOOL class type 394
  - unassigned to logical partitions 245
  - unassigned, modifying definition of 261
  - user visibility 401
  - visibility 399
- service object
  - specifying replicates 539
  - specifying replicates for 539
- Service Object Properties dialog 247, 260
- Service type (logging filter) 186

- Session dialog duration 409
  - error handling 410
  - events and 411
  - state information 410
  - transactions and 410
- Session ID (field)
  - capture and playback 168
- SetViewNodes method 115
- Shared object 335–343
  - automatic locking 336
  - cloning 343
  - common mutex deadlock 339
  - distributed 342
  - distributed mutex deadlock 340
  - events and 337
  - IsShared attribute and 335
  - mutex 336
  - nested method invocations 339
  - transactional locking and 347
- Shared partition 258
- Shared property 326
- ShowApp command (Fscript) 481
- simple data types
  - getting information about 504
- Size partnership
  - as portability tool 135–136
  - commands 135
  - minimum height/width 135
  - minimum rows/columns for character fields 129
  - using 145
- Size policy
  - and grid fields 137, 138
  - for character fields 129
  - for fields 136
  - for graphics 130
  - minimum height/width 143
- Size Policy property 95
- Source Code Management 527
  - file export formats 533
  - overview 528
  - scm utility 531
- SourceCodeManager Library 527
- SourceCodeManager.SCM class methods 531
- SP\_MT\_AUDIT 186
- SP\_MT\_CONFIGURATION 186
- SP\_MT\_DEBUG 186
- SP\_MT\_ERROR 186
- SP\_MT\_PERFORMANCE 186, 219
- SP\_MT\_RESOURCE 186
- SP\_MT\_SECURITY 186
- SP\_ST\_USER\* 187, 219
- SSL 581
  - about 550, 581
  - classes 584
  - code examples 589
  - configuring secure sessions 575
  - creating a root certificate 586
  - secure HTTP sessions 550
  - services 583
- Standard output file 210
- Standard partition 270
- Start class 328
- Start task statement
  - transactional locking and 355
  - windows and 61
- StartDocument() method 608
- StartElement() method 608
- State information
  - defined 403
  - dumping for widget 167
  - message dialog duration 406
  - session dialog duration 410
  - transaction dialog duration 408
- State property 95
- Status-line help 155
  - for menu widgets 159
  - for palette lists 158
  - menu widgets 159
- StatusText attribute 148
- Step 375
- Submenu, prefabricated 132
- Supplier Plans command
  - for libraries 384
- Synchronous processing
  - HTTPSupport library 556
- SYNTAX\_ERR 601



## T

- Tab folder 7171–84
  - about 71
  - adding a tab page 76
  - AfterTabSelect event 83
  - creating dynamically 81
  - creating in Window Workshop 73
  - deleting a tab page 78
  - editing 76
  - editing tab label 79
  - Group Into TabFolder command 74
  - Header Style property 79
  - Layout Policy property 79
  - loading data into 82
  - merging two tab folders 84
  - New TabFolder command 73
  - properties dialog for 81
  - reordering tab pages 78
  - selecting 80
  - SetPages method 81
  - tab labels 74, 75
- Tab label 74, 75
  - editing 79
- Tabfolders sample application 644
- Task, mutex and 337
- Tclient command 164
- Test method 217
- TestClient utility 163–167
  - exiting 166
  - setting refresh interval 166
  - starting 164
  - Test Client window 165
  - testing the client 166
- Testing
  - applets 443
  - AutoTester project 167
  - dynamic class loading 389
  - nomadic client application 451
  - TestClient utility 163
- Text class 600
- Text.splitText() method 599
- TextData datatype 601
- Thread Package property 267, 295
- TimeIt sample application 647
- TimeItV1-4 sample applications 463, 645
- TOOL class service object 394
  - setting attribute values 413
- Transaction
  - message dialog duration and 406
  - propagation to remote partition 356
  - session dialog duration 410
  - transaction dialog duration 409
- Transaction dialog duration 407
  - error handling 408
  - events and 409
  - state information 408
  - transactions and 409
- Transactional deadlock 351
- Transactional lock
  - definition 347
  - multiple tasks and 355
  - mutex and 347
  - non-shared objects and 349
  - read locks 348
  - write lock 348
- Transactional logging
  - events and 346
  - non-transactional objects and 345
  - scalars and 345
- Transactional object 343–356
  - cloning 356
  - events and 346
  - IsTransactional attribute 343
  - lock promotion deadlock 352
  - locking in nested transaction 355
  - logging for 345
  - not in transaction 356
  - read lock 349
  - shared 347
  - start task statement and 355
  - transactional deadlock 351
  - transactional lock 347
  - write lock 350
- Transactional property 326
  - IsTransactional attribute and 344
  - nested objects and 344
  - propagating transaction to remote partition 356
- trc message type 186

Tree view field 101115–125  
 allowing dragging 118  
 compared to outline field 115  
 controls 90, 118  
 creating in Window Workshop 120  
 data for 116  
 displaying root node 90, 118  
 interacting with 116  
 properties 118  
 properties dialog 120  
 providing data for 121  
 root node 121  
 row highlights 119  
 scroll policy 119  
 scrollbars 119

## U

Uninstall command 319  
 Upgrade approaches  
 changes allowed during 459  
 choosing among 457  
 compared 454–455  
 issues 456  
 User interface 57–68  
 dialog boxes 63  
 opening windows 57  
 TestClient utility 167  
 User interface, making portable  
 alignment and spacing tools 134  
 and grid fields 134  
 dialog boxes 131  
 fonts 129–130  
 graphics 130  
 image resolution 130  
 menus 132  
 palette lists 131  
 picture buttons 131  
 prefabricated submenus 132  
 size policies 143  
 text label sizing 129  
 widget differences 128  
 User visibility 401  
 UUID (universally unique identifier) for library 291

## V

Version, class 460  
 removing 496  
 setting 492  
 using 492  
 VertPixelsPerInch attribute 131  
 Virtual attribute, in interface 368  
 Visibility  
 service objects and 399

## W

Widget  
 state information, dumping 167  
 Widget differences  
 dialog boxes 131  
 fonts 129–130  
 list views 128  
 menus 132  
 Widgets 159  
 Width property 95  
 WidthInPixels attribute 131  
 Window  
 active 59  
 child events 65  
 close window event chain 68  
 closing 57  
 creating window object 59  
 Display method for 58  
 displaying 59  
 event loop for 64  
 field selection event chain 67  
 inactive 59  
 inherited 49  
 input focus event chain 65  
 invoking Display method 60  
 kinds 47  
 multitasking for concurrent 61  
 nested 51, 61  
 opening 57  
 page template 54  
 Parent attribute 61  
 PurgeEvents method 67

Window (*continued*)  
 specifying the border 143  
 start task statement and 61

Window class  
 FileOpenDialog method 131  
 FileSaveDialog method 132  
 MessageDialog method 132  
 PrintDialog method 132  
 QuestionDialog method 132

Window Workshop  
 Group Into TabFolder command 74  
 List View Properties dialog 94, 110  
 New ListView command 108  
 New TabFolder command 73  
 New TreeView command 120  
 Tree View Properties dialog 120

Windows 95, automatic compilation and 277

WindowSystem class  
 HeightInPixels attribute 131  
 HorzPixelsPerInch attribute 131  
 VertPixelsPerInch attribute 131  
 WidthInPixels attribute 131

WinHelp method 148

Write lock 350

## X

XML  
 about 518  
 source documents 519

XML events  
 defined 605

XML Namespaces  
 defined 594, 604  
 support for 601

XML Parsing  
 event-based 604

XML parsing  
 tree-based 595

XMLFilter interface 614

XMLFilterImpl class 614

XMLReader interface 614

XMLReader.setErrorHandler() method 610

XMLReader.setFeature() method 610

XMLReaderFactory class 614

XMLSAX2  
 filters 609  
 support for namespaces 610

XSLT processor 517  
 about 521  
 classes 526  
 features 517  
 in TOOL application 522  
 stylesheets 520  
 using protocol handlers 523  
 XSL transformations 519

