Jade Software Corporation Limited cannot accept any financial or other responsibilities that may be the result of your use of this information or software material, including direct, indirect, special or consequential damages, or loss of profits. There are no warranties extended or granted by this document or software material.

You should be very careful to ensure that the use of this software material and/or information complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Jade Software Corporation Limited.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Copyright © 2018 Jade Software Corporation Limited.

All rights reserved.

JADE is a trademark of Jade Software Corporation Limited. All trade names referenced are the service mark, trademark, or registered trademark of the respective manufacturer.

For details about other licensing agreements for third-party products, you must read the JADE ReadMe.txt file.
Contents

Before You Begin ........................................................................................................... xvi
  Who Should Read this Reference ........................................................................... xvi
  What’s Included in this Reference ......................................................................... xvi
  Conventions .............................................................................................................. xvii
  Related Documentation ............................................................................................ xvii

Chapter 1  JADE Language Reference ....................................................................... 19
  Concepts of the JADE Language ............................................................................. 19
  JADE Language Notation ......................................................................................... 20
  JADE Language Syntax ............................................................................................ 20
    Method Name ........................................................................................................ 21
    Parameters ............................................................................................................ 21
    Pseudo Types ....................................................................................................... 23
      InstanceType ....................................................................................................... 23
      KeyType ............................................................................................................... 24
      MemberType ....................................................................................................... 24
      ParamListType ..................................................................................................... 24
      SelfType .............................................................................................................. 24
      PropertyListType ................................................................................................. 25
  Passing Variable Parameters to Methods ................................................................ 25
    Variable Parameter List Considerations ................................................................ 25
      De-referencing Variable Parameters in a JADE Method ........................................ 25
      Usage Restrictions ............................................................................................. 26
    Variable Parameter Lists Cannot be Amended ..................................................... 26
    Variable Parameter Lists Cannot be Passed to External Functions ...................... 26
    No External (C++) Representation of Variable Parameter Lists ............................ 26
    Defining and Using External Dictionary Methods ................................................ 26
    Using Generic JADE Dictionary Methods ............................................................ 26
    Return Type ......................................................................................................... 27
    Method Options .................................................................................................... 27
      abstract Option ...................................................................................................... 28
      clientExecution Option ....................................................................................... 29
      condition Option ................................................................................................. 29
      conditionSafe Option ......................................................................................... 30
      final Option ......................................................................................................... 30
      lockReceiver Option ........................................................................................... 30
      mapping Option .................................................................................................... 31
      notImplemented Option ....................................................................................... 32
      partitionMethod Option ....................................................................................... 32
      protected Option .................................................................................................. 33
      serverExecution Option ...................................................................................... 34
      subschemaCopyFinal Option ................................................................................ 34
      subschemaFinal Option ....................................................................................... 35
      subschemaHidden Option .................................................................................... 35
      typeMethod Option .............................................................................................. 35
      unitTest Option ................................................................................................... 35
      unitTestAfter Option ......................................................................................... 36
      unitTestAfterAll Option ..................................................................................... 36
      unitTestAfterClass Option .................................................................................. 36
      unitTestBefore Option ....................................................................................... 37
      unitTestBeforeAll Option ................................................................................. 37
      unitTestBeforeClass Option ............................................................................... 37
      unitTestIgnore Option ......................................................................................... 38
      updating Option ................................................................................................. 38
      webService Option ............................................................................................. 39
    Server and Client Execution Restrictions ............................................................. 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant Declaration</td>
<td>40</td>
</tr>
<tr>
<td>Variable Declaration</td>
<td>40</td>
</tr>
<tr>
<td>JADE Instructions</td>
<td>41</td>
</tr>
<tr>
<td>Epilog</td>
<td>41</td>
</tr>
<tr>
<td>Using Comments in Your Methods</td>
<td>41</td>
</tr>
<tr>
<td>Type Methods</td>
<td>42</td>
</tr>
<tr>
<td>Invoking a Type Method</td>
<td>44</td>
</tr>
<tr>
<td>Accessing Meta Data</td>
<td>45</td>
</tr>
<tr>
<td>Accessing a Property, Constant, or Method in a Metaschema</td>
<td>45</td>
</tr>
<tr>
<td>Meta Data Access Restriction</td>
<td>45</td>
</tr>
<tr>
<td>Accessing a Meta-Data Class in an Imported Package</td>
<td>46</td>
</tr>
<tr>
<td>JADE Instructions, Expressions, and Assignments</td>
<td>46</td>
</tr>
<tr>
<td>Instructions</td>
<td>46</td>
</tr>
<tr>
<td>Expressions</td>
<td>46</td>
</tr>
<tr>
<td>Literals</td>
<td>47</td>
</tr>
<tr>
<td>StringUtf8 Literals</td>
<td>47</td>
</tr>
<tr>
<td>Types of Numeric Literals</td>
<td>48</td>
</tr>
<tr>
<td>Hexadecimal Literals</td>
<td>48</td>
</tr>
<tr>
<td>Constants</td>
<td>49</td>
</tr>
<tr>
<td>System Variables</td>
<td>50</td>
</tr>
<tr>
<td>app</td>
<td>50</td>
</tr>
<tr>
<td>appContext</td>
<td>50</td>
</tr>
<tr>
<td>currentSchema</td>
<td>50</td>
</tr>
<tr>
<td>currentSession</td>
<td>50</td>
</tr>
<tr>
<td>exception</td>
<td>51</td>
</tr>
<tr>
<td>global</td>
<td>51</td>
</tr>
<tr>
<td>method</td>
<td>51</td>
</tr>
<tr>
<td>node</td>
<td>52</td>
</tr>
<tr>
<td>process</td>
<td>52</td>
</tr>
<tr>
<td>rootSchema</td>
<td>52</td>
</tr>
<tr>
<td>self</td>
<td>52</td>
</tr>
<tr>
<td>selfType</td>
<td>53</td>
</tr>
<tr>
<td>system</td>
<td>53</td>
</tr>
<tr>
<td>Expressions with Operators</td>
<td>53</td>
</tr>
<tr>
<td>Unary Operators</td>
<td>54</td>
</tr>
<tr>
<td>Binary Operators</td>
<td>54</td>
</tr>
<tr>
<td>Arithmetic Binary Operators</td>
<td>54</td>
</tr>
<tr>
<td>Boolean Binary Operators</td>
<td>55</td>
</tr>
<tr>
<td>Relational Binary Operators</td>
<td>56</td>
</tr>
<tr>
<td>String Concatenation Operator</td>
<td>58</td>
</tr>
<tr>
<td>Substring Operators</td>
<td>59</td>
</tr>
<tr>
<td>Using Subscripts in Dictionaries and Arrays</td>
<td>60</td>
</tr>
<tr>
<td>Operator Precedence</td>
<td>60</td>
</tr>
<tr>
<td>Type Casts</td>
<td>61</td>
</tr>
<tr>
<td>Converting Primitive Types</td>
<td>61</td>
</tr>
<tr>
<td>Binary Primitive Type Conversions</td>
<td>67</td>
</tr>
<tr>
<td>Type Guard Expression</td>
<td>68</td>
</tr>
<tr>
<td>Using Type Guards</td>
<td>68</td>
</tr>
<tr>
<td>Assignments</td>
<td>69</td>
</tr>
<tr>
<td>Assignment Compatibility</td>
<td>69</td>
</tr>
<tr>
<td>Issues and Side Effects</td>
<td>70</td>
</tr>
<tr>
<td>Object References versus Primitive Values</td>
<td>70</td>
</tr>
<tr>
<td>Epilog</td>
<td>71</td>
</tr>
<tr>
<td>Iterative Instructions</td>
<td>72</td>
</tr>
<tr>
<td>while Instruction</td>
<td>72</td>
</tr>
<tr>
<td>Terminating a while Iteration</td>
<td>73</td>
</tr>
<tr>
<td>foreach Instruction</td>
<td>73</td>
</tr>
<tr>
<td>Iterating over a Range of Numeric Values</td>
<td>74</td>
</tr>
<tr>
<td>Examples of Numeric Value Iteration</td>
<td>76</td>
</tr>
<tr>
<td>Terminating a foreach Iteration</td>
<td>76</td>
</tr>
<tr>
<td>where Clause Optimization</td>
<td>77</td>
</tr>
</tbody>
</table>
## Chapter 2 Notifications

### Overview

- **Using Notifications** .......................................................... 114
- **System Notifications** ...................................................... 115
- **User Notifications** .......................................................... 115

**Beginning a Notification** .................................................. 115

**Example** ............................................................................ 118

**Receiving a Notification** .................................................... 119

- **Receiving System Notifications** ........................................ 119
  - **eventType** ............................................................ 119
  - **theObject** ............................................................ 119
  - **eventTag** ............................................................ 120
  - **System Notification Examples** ...................................... 120
- **Receiving User Notifications** ........................................... 120
  - **eventType** ............................................................ 121
  - **theObject** ............................................................ 121
  - **eventTag** ............................................................ 121
  - **userInfo** ............................................................. 121
  - **User Notification Examples** ......................................... 121

**Ending a Notification** ....................................................... 122

**Using Events** ...................................................................... 124
Chapter 3  Exceptions ......................................................... 128
Overview ................................................................. 128
  Error Handling Policy ................................................ 129
  System Exceptions .................................................. 129
  User Exceptions ..................................................... 130
Handling Exceptions .................................................... 130
  Automatic Updates and Exception Handling ...................... 132
Dealing with System Exceptions ...................................... 132
  Default Exception Handler ....................................... 134
  Debugging the Method Call Stack ................................ 135
Creating and Arming an Exception Handler ......................... 136
  Creating an Exception Handler ................................... 136
  Examples of Defining an Exception Handler ..................... 140
Arming an Exception Handler ......................................... 141
  Examples of Arming an Exception Handler ....................... 142
Disarming a Local Exception Handler ............................... 143
Disarming a Global Exception Handler ............................. 143
Viewing the Exception Handler Stack ............................... 143
Using Exception Handlers in Non-JADE Applications .............. 143
  Using the raise_ Method ............................................ 144
  Using the raise exception Instruction ........................... 145
Interpreting Exceptions ............................................... 146
  Monitoring Forms to Identify an Exception Dialog .............. 149

Chapter 4  Collections Behavior and Tuning .................................. 150
Overview ................................................................. 150
  Tuning the Btree Load Factor ...................................... 150
  Tuning the Block Size at Collection Type Level .................. 151
  Tuning the Block Size at Instance Level .......................... 151
Collection Behavior .................................................... 151
  Precedence of Block Size and Load Factor Values ............... 151
    Class-Level Specification ......................................... 151
    Instance-Level Specification ................................... 152
  Tuning Collections at Development Time .......................... 152
  Using Methods to Dynamically Tune Collections at Run Time ... 152
Analyzing a Collection ............................................... 153
  Performance Considerations ..................................... 153

Chapter 5  Creating Your Own Control Classes .................................. 154
Overview ................................................................. 154
  Subclassing a JADE Control Class ................................ 154
    Selecting Your Design Time Properties ......................... 156
      Controlling the Display in the Properties Dialog ............ 157
      Defining a List Value .......................................... 158
    Adding Properties to Your Subclassed Control ............... 159
    Adding Methods to Your Subclassed Control ..................... 159
    Examples .......................................................... 160

Chapter 6  JADE Locking .................................................. 163
Overview ................................................................. 163
  Locking Objects ..................................................... 166
    Exclusive Locks ................................................... 167
    Shared Locks ...................................................... 167
    Reserve Locks ..................................................... 167
Contents

Chapter 7 Using the Database Administration Framework ........................................... 178

Overview .................................................................................................................. 178
Using the Supplied Database Administration Framework ....................................... 179
  DbFile Class Constants, Properties, and Methods Summary ..................................... 179
  JadeDatabaseAdmin Class Constants and Methods Summary ................................ 181
Simple Scripted Database Backup .............................................................................. 182
  backupDatabase Method ....................................................................................... 182
Developing Backup Applications .............................................................................. 183
  DbAdmin Class Partial Specification ...................................................................... 183
  BackupManager Class Partial Specification .......................................................... 184
  BackupDatabaseDlg Class Partial Specification .................................................... 184
Initialization ............................................................................................................. 185
  BackupDatabaseDlg::initialise Method .................................................................. 185
  BackupManager Constructor ................................................................................... 186
  BackupManager::initialise Method ....................................................................... 186
Processing Backups Asynchronously ....................................................................... 186
  BackupDatabaseDlg::doBackup Method ................................................................ 187
  DbAdmin::backupDatabaseProcess method ............................................................ 187
  BackupManager::backupHandler Method ................................................................ 187
Notification and Progress Dialog Handling .............................................................. 188
  BackupDatabaseDlg::userNotify Method ................................................................ 188
  BackupDatabaseDlg::updateProgress Method ....................................................... 189
  BackupDatabaseDlg::cancelBackup Method ............................................................ 189
Backup Transactions and Partial Database Backups .................................................. 190
  BackupManager::backupDataFiles Method ............................................................. 190
Exception Handling ................................................................................................. 191
  BackupManager::backupException Method ........................................................... 191
Final Housekeeping Tasks ......................................................................................... 191
  BackupDatabaseDlg::finalise Method ................................................................... 191
  BackupDatabaseDlg::unload Method .................................................................... 192
  BackupManager::finalise Method ......................................................................... 192
  BackupManager::getDroppedFileText Method ....................................................... 192
  BackupManager Destructor Method ...................................................................... 192
Using the Online Database Backup Service ............................................................. 193
Creating and Using the Backup Database Dialog in User Application Logic ............. 193
Using the Supplied Online Backup Dialog .............................................................. 193

Update Locks ........................................................................................................... 168
Exception Handling ................................................................................................. 168
Cache Concurrency .................................................................................................. 169
  Defining Volatility State ....................................................................................... 170
  Determining Object Volatility ............................................................................. 170
Stable Objects ........................................................................................................ 170
  Shared Locks on Stable Objects ......................................................................... 171
  Reserve and Session Duration Locks on Stable Objects ....................................... 171
  Exclusive Locks on Stable Objects ...................................................................... 172
  Server Applications, Single User Mode, and Server Execution Methods ............... 172
Frozen Objects ........................................................................................................ 172
  Shared and Reserve Locks on Frozen Objects ..................................................... 172
  Exclusive Locks on Frozen Objects ..................................................................... 172
  Upgrading Frozen Objects ................................................................................... 172
Freezing and Thawing Schema Files ....................................................................... 173
Handling Deadlocks .............................................................................................. 174
Lock Handling by the JADE Server ........................................................................ 174
Upgrading and Downgrading Locks ......................................................................... 174
  Lock Durations .................................................................................................... 174
  Lock Types .......................................................................................................... 174
Unlocking Objects .................................................................................................. 176
Using Object Editions .............................................................................................. 177
Chapter 8 Using Packages .......................................................... 195
  Overview .............................................................................. 196
  Packages Usage Example ..................................................... 196
    Exported Package Example ............................................... 199
    Imported Package Example .............................................. 200
    Locally Defined Imported Class Method Example .............. 201
  Exported and Imported Packages ......................................... 202
    Exported Package Overview ............................................. 202
    Imported Package Overview ............................................ 203
      Extending Imported Classes ........................................... 204
      Implicit Inheritance from Nearest RootSchema Subschema Copy Class .................. 204
    Method Dispatch ................................................................ 205
    Example of Method Dispatch Using the importMethod Instruction .......... 208
    Preventing Multiple Imports of the Same Class ................. 208
    Persistent Imported Class Instances ................................. 209
  Environmental Objects ....................................................... 209
  Locales .............................................................................. 210
  Timer, Notification, and Control Event Methods ................... 210
  Extracting and Loading Package Schemas ............................ 210
  Exporting a Package Using the Export Package Definition Wizard ................................................. 211
    Using the Packages Menu for Exported Packages .................. 212
    Using the Export Package Definition Wizard to Define a Package .......................................................... 212
      Defining the Package .................................................... 213
      Creating a Working Set of Classes ................................... 214
      Selecting Classes and Features ...................................... 216
      Selecting Interfaces ..................................................... 217
      Changing Attributes .................................................... 219
    Changing a Package by Using the Export Package Definition Wizard ................................................. 221
    Browsing and Maintaining an Exported Package .................. 221
      Maintaining a Class in the Export Package Browser ........... 222
      Maintaining a Property in the Export Package Browser ........ 224
      Locating a Class .......................................................... 226
    Displaying References to an Exported Package ..................... 228
  Importing a Package .......................................................... 228
    Using the Packages Menu for Imported Packages .................. 228
    Adding an Imported Package to Your Schema ..................... 229
    Changing an Imported Package ....................................... 231
    Browsing and Maintaining an Imported Package .................. 232
  Stub Packages ................................................................. 233
    Extracting a Package ..................................................... 233
    Removing a Package ..................................................... 233
    Switching Application Contexts When Invoking a Method .......... 234
    Calling User Methods from Packages .................................. 235
    Package Callback Examples ............................................. 237
    Summary of Restrictions when Using Packages .................... 238

Chapter 9 Using Skins to Enhance JADE Applications .................. 240
  Overview .............................................................................. 240
    JADE Development Environment Skin Classes .................... 241
    JadeSkinRoot Class ......................................................... 243
    JadeSkinEntity Class and Subclasses .................................. 243
    JadeSkinApplication Class .............................................. 244
    JadeSkinArea Class ....................................................... 244
    JadeSkinMenu Class ....................................................... 244
    JadeSkinWindow Class ................................................... 247
    JadeSkinForm Class ....................................................... 248
    JadeSkinControl Class ................................................... 252
    JadeSkinBaseControl Class ............................................. 254
    JadeSkinBrowseButtons Class ......................................... 254
Chapter 10    Synchronized Database Service (SDS) Development Considerations ................................. 271

Overview ............................................................................................................................................. 271
Using the JadeDatabaseAdmin Class SDS Methods .......................................................................... 272
JadeDatabaseAdmin Methods .................................................................................................................. 272
Returning Information about SDS Entities .......................................................................................... 274
SDSPrimary Dynamic Object Type ......................................................................................................... 274
SDSSecondary Dynamic Object Type .................................................................................................... 275
SDSSecondaryProxy Dynamic Object Type .......................................................................................... 280
SDSTransaction Dynamic Object Type .................................................................................................. 281

SDS Global Constants .......................................................................................................................... 282
JadeDynamicObjectNames Category Global Constants ......................................................................... 282
JadeDynamicObjectTypes Category Global Constants ............................................................................. 283
JadeErrorCodesSDS Category Global Constants ..................................................................................... 283
SDSConnectionState Category .................................................................................................................. 283
Chapter 11 Building Web Services Applications

Overview .............................................................................................................. 296
SOAP-Based Web Services .................................................................................. 297
JADE Web Service Provider ............................................................................. 298
JADE Web Service Consumer ......................................................................... 298
SOAP Versions .................................................................................................. 299
Defining the Classes, Properties, and Methods for a Web Service ................. 300
Creating a Web Service Class .......................................................................... 300
Creating a SOAP Header Class ....................................................................... 302
Creating Web Service Methods ...................................................................... 303
Specifying Your Web Services Options ............................................................ 304
Using the Web Service Exposure Wizard .......................................................... 305
Adding a Web Service Exposure Definition ...................................................... 306
Selecting Classes and Options for Your Web Service Exposure ...................... 307
Selecting Properties for Inclusion in Your Web Service Exposure ................. 309
Displaying a Hierarchical Web Service Exposure Browser ........................... 311
Toggling the Composite View of the Hierarchy Exposure Browser .................. 313
Changing a Web Service Exposure Definition .................................................. 314
Removing a Web Service Exposure Definition ............................................... 314
Extracting a Web Service Exposure .................................................................. 315
Generating JavaScript for a Web Service Exposure ......................................... 315
Defining a Web Services Provider Application ............................................... 316
Generating Web Services Description Language (WSDL) ............................... 318
Setting Up Version Control .............................................................................. 319
Testing the Web Services Interface .................................................................. 319
Web Service Provider Example ......................................................................... 324
Using the Generated WSDL .............................................................................. 330
Defining a Web Service Consumer Application ............................................. 332
Using the Web Service Consumer Wizard to Add a Web Service Consumer ...... 333
Specifying WSDL File and Consumer Names ................................................... 333
Defining Your Web Service Consumer Mappings ............................................ 336
Using the Imported Web Service Consumer ................................................... 338
Example of Running a Web Service Synchronously ..................................... 338
Example of Running a Web Service Asynchronously ..................................... 339
Displaying a Hierarchical Web Service Consumer Browser ......................... 340
Reloading a Web Service Consumer ................................................................. 340
Removing a Web Service Consumer ................................................................. 341
Extracting a Specific Web Service Consumer .................................................. 341
Chapter 12  Processing XML Documents

Overview ............................................................................................................. 369
XML Syntax ......................................................................................................... 370
Jade XML Tree Model .......................................................................................... 370
XML Tree Architecture ......................................................................................... 371
Using the XML Tree Model .................................................................................. 373
Parsing an XML Tree Document ......................................................................... 373
Processing Extra Whitespace ................................................................................ 374
Creating XML Tree Documents ......................................................................... 375
Retrieving Information from XML Tree Documents ............................................ 376
Traversing XML Documents ................................................................................ 377
Modifying XML Tree Documents ........................................................................ 378
Writing XML Tree Documents ............................................................................ 379
Handling XML Tree Exceptions ........................................................................... 379
Chapter 15  Using the Messaging Framework ................................................................. 424
  Overview ................................................................................................................. 424
  Components of the Messaging Framework .............................................................. 425
  Messaging Terminology ............................................................................................ 425
  Concept of Queued Messaging ................................................................................ 426
  Classes in the Messaging Framework ....................................................................... 426
  Examples of the Use of the Messaging Framework .................................................... 426
  Message Payload ...................................................................................................... 428
    Unstructured Data .................................................................................................. 429
    Structured Data ...................................................................................................... 429
  Specific Transports .................................................................................................... 430
    JADE Messaging Transport ..................................................................................... 431
    Restricting Access to Queues ................................................................................. 431
    WebSphere Messaging Transport ........................................................................... 432
      Opening a WebSphere MQ Queue ....................................................................... 433
      Name Lengths ...................................................................................................... 433
      Message Format Property ..................................................................................... 433
      Message messageID Property .............................................................................. 434
      WebSphere-Specific Errors ................................................................................... 434
  Generic Message Example ........................................................................................ 434
  Server Application ..................................................................................................... 434
  Client Application ..................................................................................................... 437

Chapter 16  Using Asynchronous Method Calls .............................................................. 440
  Overview .................................................................................................................. 440
  Asynchronous Worker Applications .......................................................................... 441
  Making an Asynchronous Method Call ...................................................................... 441
    Preparing for an Asynchronous Method Call .......................................................... 441
    Invoking the Asynchronous Method Call ................................................................. 442
    Timeout .................................................................................................................. 442
    Waiting for Asynchronous Method Calls to Complete ............................................. 442
    Obtaining Results from an Asynchronous Method Call .......................................... 444
  Example of Using Asynchronous Method Calls in a Text Search .............................. 445

Chapter 17  Using the JADE Testing Framework ............................................................. 449
  Overview .................................................................................................................. 449
  JADE Unit Testing .................................................................................................... 450
  Writing Unit Tests ..................................................................................................... 450
    Generating a Web Service Consumer Unit Test Class and Stub Methods ................. 453
    Running Unit Tests ................................................................................................. 460
Running Unit Tests from the JADE Development Environment .................................................. 460
Using the Unit Test Runner Form .................................................................................................. 461
Performing File Menu Command Actions .................................................................................... 462
Increasing the Font Size .................................................................................................................. 462
Decreasing the Font Size ................................................................................................................. 462
Displaying the Call Stack for Failed Methods .............................................................................. 463
Displaying the Call Stack for Exceptions ...................................................................................... 463
Displaying the Call Stack for Unexpected Exceptions ................................................................ 464
Enabling Code Coverage .................................................................................................................. 464
Reporting Code Coverage Results ................................................................................................. 464
Viewing Code Coverage Results ...................................................................................................... 465
Starting the Collection of Unit Test Results .................................................................................. 465
Outputting Unit Test Results to a File ............................................................................................ 465
Exiting from the Unit Test Runner Form .......................................................................................... 465
Performing View Menu Command Actions .................................................................................. 466
Including Passed Tests in the Results Pane .................................................................................... 466
Selecting Passed Tests in the Select Tests Pane ............................................................................ 466
Selecting Failed Tests in the Select Tests Pane ............................................................................ 466
Selecting Not Run Tests in the Select Tests Pane .......................................................................... 467
Refreshing the Unit Test Results ...................................................................................................... 467
Running Unit Tests in Batch Mode .................................................................................................. 467
Running Batch Mode Unit Tests that Require GUI Components .................................................... 470
Running Unit Tests from an Application .......................................................................................... 471
Listening for Unit Test Results ........................................................................................................ 472
Example of Using the JADE Unit Test Framework ........................................................................ 474
Code Coverage .................................................................................................................................. 478
Collecting Code Coverage Results .................................................................................................. 478
Viewing Code Coverage Results ...................................................................................................... 479
Code Coverage Results Browser ...................................................................................................... 480
Using Code Coverage Menus ............................................................................................................ 485
Performing File Menu Command Actions .................................................................................... 485
Loading Code Coverage Files ........................................................................................................... 485
Loading and Merging Coverage for Methods ................................................................................ 486
Loading and Replacing Coverage for Methods ............................................................................. 486
Clearing Session Information ........................................................................................................... 487
Saving Code Coverage Results in CSV Format ............................................................................ 487
Saving Methods not in the Code Coverage .................................................................................... 487
Calculating as a Percentage the Total Methods Executed ............................................................. 488
Exiting From the Code Coverage Results Browser ........................................................................ 488
Performing View Menu Command Actions .................................................................................. 488
Viewing a Schema, Type, Method Hierarchy .................................................................................. 488
Viewing Schemas Only .................................................................................................................... 489
Viewing Types Only .......................................................................................................................... 489
Viewing Methods Only .................................................................................................................... 489
Sorting By Name ............................................................................................................................... 489
Sorting By Total Blocks ................................................................................................................... 489
Sorting By Covered Blocks .............................................................................................................. 490
Sorting By Covered Blocks % .......................................................................................................... 490
Sorting in Ascending Order ............................................................................................................. 490
Sorting in Descending Order ............................................................................................................ 490
Displaying a List of Entities that are not Covered ......................................................................... 490

Chapter 18 Tracking Methods ......................................................................................................... 491
Overview .......................................................................................................................................... 491
Enabling Tracking ............................................................................................................................. 491
Tracking a Method ............................................................................................................................ 492
Helper Methods for Obtaining Method Tracking Information ........................................................ 492
Examples of Using Method Tracking ............................................................................................... 493
# Contents

## Chapter 19   Tracing Transactions  ................................................................. 495  
Overview ......................................................................................... 495
Starting and Stopping Transaction Tracing  .......................................................... 495
Registering Properties for Transaction Tracing ...................................................... 495
Analyzing Transaction Trace Information .............................................................. 496

## Chapter 20   Partitioning Database Files ............................................................... 498
Overview ......................................................................................... 499
Partitioning Example .............................................................................. 499
State of a Partition .............................................................................. 500
Database Partitioned File Structures ............................................................... 502
Partition Control File ............................................................................ 502
Partition Index File .............................................................................. 503
Partition Data Files .............................................................................. 503
Partition Index, Modulus, and Creation Window ................................................ 503
Preparing to Partition a Database File ............................................................ 504
Empty Database File .............................................................................. 505
Non-Empty Database File ........................................................................ 505
Flexibility in Partitioning Database Files .......................................................... 505
Partition Location .................................................................................. 505
Partitioning by Period ............................................................................ 506
Automatic Partitioning ............................................................................ 507
Implementing a Partition Algorithm .............................................................. 507
Partition Methods ................................................................................ 507
Determining the Partition Index .................................................................. 508
Manual Partitioning ............................................................................... 508
Bulk Removal of Objects ......................................................................... 509
RootSchema Methods Related to Database Partitioning ........................................ 509
Class Methods .................................................................................... 509
DbFile Property and Methods .................................................................... 509
JadeDbFilePartition Properties and Methods .................................................. 510
Object Methods ................................................................................... 511
System Method .................................................................................... 511
Batch JADE Database Administration Utility Partitioning Actions ......................... 512

## Chapter 21   Runtime Data Definition .............................................................. 513
Overview ......................................................................................... 513
Runtime Dynamic Properties ..................................................................... 514
Dynamic Property Clusters ....................................................................... 514
Creating and Maintaining a Dynamic Property Cluster .................................... 515
Runtime Dynamic Properties ..................................................................... 516
Adding a Runtime Dynamic Property ........................................................... 516
Changing a Runtime Dynamic Property ....................................................... 517
Maintaining the Value of a Runtime Dynamic Property .................................... 518
Deleting a Runtime Dynamic Property ........................................................ 518
Inverse References Involving Runtime Dynamic Properties .............................. 519
Runtime Dynamic Property Restrictions ....................................................... 520
Adding User Classes at Run Time .................................................................. 521
Dynamic Objects ................................................................................... 522
DynaDictionary Objects .......................................................................... 523

## Chapter 22   Unaudited Database File Operations .................................................. 525

## Appendix A   JADE Limits .................................................................................. 526

## Appendix B   Attribute Type Storage Sizes ........................................................ 530
Before You Begin

The JADE Developer's Reference is intended as a major source of information when you are developing or maintaining JADE applications.

Who Should Read this Reference

The main audience for the JADE Developer's Reference is expected to be developers of JADE application software products.

What’s Included in this Reference

The JADE Developer's Reference has twenty-two chapters and two appendixes.

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>A reference for the JADE language</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Handling event notifications</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>Handling exceptions</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>The behavior and tuning of collections</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Creating user-defined controls</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Locking objects</td>
</tr>
<tr>
<td>Chapter 7</td>
<td>Using the supplied database administration framework</td>
</tr>
<tr>
<td>Chapter 8</td>
<td>Using packages to build and reuse common frameworks</td>
</tr>
<tr>
<td>Chapter 9</td>
<td>Using skins to enhance JADE applications</td>
</tr>
<tr>
<td>Chapter 10</td>
<td>Synchronized Database Service (SDS) development considerations</td>
</tr>
<tr>
<td>Chapter 11</td>
<td>Building distributed JADE applications using the XML Web services architecture</td>
</tr>
<tr>
<td>Chapter 12</td>
<td>Parsing, creating, navigating, manipulating, and writing XML documents</td>
</tr>
<tr>
<td>Chapter 13</td>
<td>Using object versioning in the JADE development environment</td>
</tr>
<tr>
<td>Chapter 14</td>
<td>Database reorganization</td>
</tr>
<tr>
<td>Chapter 15</td>
<td>Using the messaging framework</td>
</tr>
<tr>
<td>Chapter 16</td>
<td>Using asynchronous method calls</td>
</tr>
<tr>
<td>Chapter 17</td>
<td>Using the unit testing framework</td>
</tr>
<tr>
<td>Chapter 18</td>
<td>Executing code before and after a target method is called</td>
</tr>
<tr>
<td>Chapter 19</td>
<td>Identifying objects that are updated, created, and deleted within a transaction</td>
</tr>
<tr>
<td>Chapter 20</td>
<td>A reference on partitioning database files</td>
</tr>
<tr>
<td>Chapter 21</td>
<td>Defining runtime data</td>
</tr>
<tr>
<td>Chapter 22</td>
<td>Unaudited database file operations</td>
</tr>
<tr>
<td>Appendix A</td>
<td>A reference of JADE limits</td>
</tr>
<tr>
<td>Appendix B</td>
<td>A reference to the JADE attribute type storage sizes</td>
</tr>
</tbody>
</table>
Conventions

The JADE Developer’s Reference uses consistent typographic conventions throughout.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrow bullet ( › )</td>
<td>Step-by-step procedures. You can complete procedural instructions by using either the mouse or the keyboard.</td>
</tr>
<tr>
<td><strong>Bold</strong></td>
<td>Items that must be typed exactly as shown. For example, if instructed to type <code>foreach</code>, type all the bold characters exactly as they are printed. File, class, primitive type, method, and property names, menu commands, and dialog controls are also shown in bold type, as well as literal values stored, tested for, and sent by JADE instructions.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Parameter values or placeholders for information that must be provided; for example, if instructed to enter <code>class-name</code>, type the actual name of the class instead of the word or words shown in italic type. Italic type also signals a new term. An explanation accompanies the italicized type. Document titles and status and error messages are also shown in italic type.</td>
</tr>
<tr>
<td><strong>Blue text</strong></td>
<td>Enables you to click anywhere on the cross-reference text (the cursor symbol changes from an open hand to a hand with the index finger extended) to take you straight to that topic. For example, click on the &quot;Causing an Object Event&quot; cross-reference to display that topic.</td>
</tr>
<tr>
<td>( [ ] )</td>
<td>Indicate optional items.</td>
</tr>
<tr>
<td>( )</td>
<td>Separates alternative items.</td>
</tr>
<tr>
<td><strong>Monospaced font</strong></td>
<td>Syntax, code examples, and error and status message text.</td>
</tr>
<tr>
<td><strong>ALL CAPITALS</strong></td>
<td>Directory names, commands, and acronyms.</td>
</tr>
<tr>
<td><strong>SMALL CAPITALS</strong></td>
<td>Keyboard keys.</td>
</tr>
</tbody>
</table>

Key combinations and key sequences appear as follows.

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KEY1+KEY2</td>
<td>Press and hold down the first key and then press the second key. For example, &quot;press Shift+F2&quot; means to press and hold down the Shift key and press the F2 key. Then release both keys.</td>
</tr>
<tr>
<td>KEY1,KEY2</td>
<td>Press and release the first key, then press and release the second key. For example, &quot;press Alt+F.X&quot; means to hold down the Alt key, press the F key, and then release both keys before pressing and releasing the X key.</td>
</tr>
</tbody>
</table>

Related Documentation

Other documents that are referred to in this reference, or that may be helpful, are listed in the following table, with an indication of the JADE operation or tasks to which they relate.

<table>
<thead>
<tr>
<th>Title</th>
<th>Related to…</th>
</tr>
</thead>
<tbody>
<tr>
<td>JADE Database Administration Guide</td>
<td>Administering JADE databases</td>
</tr>
<tr>
<td>Title</td>
<td>Related to…</td>
</tr>
<tr>
<td>---------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>JADE Development Environment Administration Guide</td>
<td>Administering JADE development environments</td>
</tr>
<tr>
<td>JADE Developer's Reference</td>
<td>Developing or maintaining JADE applications</td>
</tr>
<tr>
<td>JADE Automated Test Code Generator (ATCG) Reference</td>
<td>Recording and replaying GUI actions in JADE applications</td>
</tr>
<tr>
<td>JADE Encyclopaedia of Classes</td>
<td>System classes (Volumes 1 and 2), Window classes (Volume 3)</td>
</tr>
<tr>
<td>JADE Encyclopaedia of Primitive Types</td>
<td>Primitive types and global constants</td>
</tr>
<tr>
<td>JADE External interface Developer’s Reference</td>
<td>Developing JADE applications using external interfaces</td>
</tr>
<tr>
<td>JADE Installation and Configuration Guide</td>
<td>Installing and configuring JADE</td>
</tr>
<tr>
<td>JADE Initialization File Reference</td>
<td>Maintaining JADE initialization file parameter values</td>
</tr>
<tr>
<td>JADE .NET Developer’s Reference</td>
<td>Developing applications using .NET class libraries exposed in JADE</td>
</tr>
<tr>
<td>JADE Object Manager Guide</td>
<td>JADE Object Manager administration</td>
</tr>
<tr>
<td>JADE Synchronized Database Service (SDS) Administration Guide</td>
<td>Administering JADE Synchronized Database Services (SDS), including Relational Population Services (RPS)</td>
</tr>
<tr>
<td>JADE Thin Client Guide</td>
<td>Administering JADE thin client environments</td>
</tr>
<tr>
<td>JADE Web Application Guide</td>
<td>Implementing, monitoring, and configuring Web applications</td>
</tr>
</tbody>
</table>
Chapter 1  JADE Language Reference

This chapter covers the following topics.

- Concepts of the JADE Language
- JADE Language Notation
  - JADE Language Syntax
  - Using Comments in Your Methods
  - Accessing Meta Data
- JADE Instructions, Expressions, and Assignments
- Epilogs
- Iterative Instructions
- Conditional Instructions
- Other Control Structures
- Simple Input and Output Instructions
- Exception Handling Instructions
- Transaction and Concurrency Control Instructions
- Instructions for Creating and Deleting Objects
- JADE Language Vocabulary
  - Special Symbols
  - Reserved Words

Concepts of the JADE Language

The JADE language is the programming language in which JADE methods are written.

You can write external methods using other programming languages (for example, Pascal or C++), but JADE is the language of choice, for the following reasons.

- It has a simple, regular syntax
- It provides seamless access to the JADE Object Manager and user-interface components
- It includes specialized constructs for exploiting some of the more-powerful features of the JADE environment

In JADE, procedural-style control structures and arithmetic expressions are combined with object-oriented message-passing syntax. Message-passing uses a "." (dot) notation.
JADE is a strongly typed language, in which you must explicitly state the type of each identifier. When instructions refer to methods or properties, the compiler checks that the method or property has been defined for the corresponding objects. This strong typing provides the following benefits.

- Reduced risk of runtime errors due to invalid methods being invoked
- Early error detection

The JADE language syntax makes no distinction between transient and persistent objects. The same syntax is used for creating, maintaining, and deleting transient and persistent objects.

Methods can be invoked for transient objects in the same way as they can for persistent objects.

**JADE Language Notation**

This section covers the following topics.

- JADE Language Syntax
  - Method Name
  - Parameters
  - Pseudo Types
  - Passing Variable Parameters to Methods
  - Return Type
  - Method Options
  - Constant Declaration
  - Variable Declaration
  - JADE Instructions
  - Epilog
- Using Comments in Your Methods
- Type Methods
  - Invoking a Type Method
- Accessing Meta Data

**JADE Language Syntax**

The syntax definition for JADE methods and schema files uses the Extended Backus Naur Formalism (EBNF). An EBNF specification is a sequence of syntax rules.

EBNF symbols are used to precisely and concisely specify the syntax. The symbols used in EBNF are:

- Parentheses (that is, the ( and ) symbols) group alternative terms
- The vertical bar (|) separates alternative terms
- Brackets ([ ]) identify optional expressions
- Braces ( { } ) identify expressions that can occur zero or more times
Character sequences enclosed in double quotes (""") identify terminal symbols, or keywords, of the JADE language.

An identifier is a sequence of letters and digits, beginning with a letter.

The following example shows meals that are defined with a sequence of EBNF symbols.

```plaintext
appetizer = "artichoke" | "oysters"
dessert = "ice cream" | "fruit"
fruit = "apple" | "orange" | "pear"
meat = "beef" | "lamb" | "fish"
vegetable = "broccoli" | "carrots" | "peas"
meal = [appetiser] meat ("potatoes" | "rice") {vegetable} [dessert]
```

Examples of meals defined by these rules are:

- beef potatoes
- artichoke fish rice peas broccoli ice cream
- lamb rice carrots carrots peas broccoli pear
- oysters beef rice orange

A JADE method has the following syntax.

```plaintext
method-name ([parameters]) [[: return-type] [method-options];
[constants
  constant-declarations]
[vars
  variable-declarations]
begin
  JADE-instructions
  [epilog
    epilog-instructions]
end;
```

**Method Name**

The *method-name* is the name of the method defined by using the Name text box in the Jade Method Definition dialog. (For details, see "Defining and Compiling JADE Methods and Conditions", in Chapter 4 of the JADE Development Environment User’s Guide.) The method name must start with a lowercase letter.

**Parameters**

The *parameters* are a list of parameters for the method. A parameter declaration has the following syntax.

```plaintext
name[, name, name...] : type [usage];
```

The *usage* value can be **constant**, **input**, **io** (input-output), or **output**. If you do not specify a usage, a default of **constant** is assumed.

The parameter usages are as follows.

- For **constant** and **input** parameters, the parameter value specified in the method call is passed to the corresponding parameter in the called method.

  **Note** You cannot assign to an **input** parameter.

- For **output** parameters, the value is passed in the reverse direction, from the parameter of the called method...
back to the corresponding parameter of the caller.

This copying back of the parameter value occurs when the called method returns, following any epilog that may be defined.

- For **io** parameters, the parameter value is passed in both directions, as follows.
  - From the caller to the called method when the method begins execution
  - From the called method back to the caller when it returns, following any epilog that may be defined

You can define parameters only for conditions that are not constraints, and the parameters must be constant parameters. (For more details, see "Adding Conditions to Classes or Primitive Types", in Chapter 4 of the JADE Development Environment User's Guide, and "condition Option", later in this chapter.)

For details about promoting a local variable to be a parameter of the current method, see "Promoting a Local Variable to be a Parameter of the Current Method", in Chapter 4 of the JADE Development Environment User's Guide.

As the name implies, **constant** parameters cannot be updated or modified in any way within the body of the method. Specifically, the following restrictions apply to **constant** parameters.

- You cannot assign to the parameter.
- You cannot assign to properties of the parameter object.
- Updating methods cannot be invoked for the parameter.
- Updating methods cannot be invoked for properties of the parameter object.

The following table lists the use of a primitive type parameter in a method.

<table>
<thead>
<tr>
<th>Usage</th>
<th>Value Passed into Method</th>
<th>Parameter Value Can Be Updated in Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>input</td>
<td>Yes</td>
<td>Cannot be updated by assignment, but can be updated within an updating primitive type method by using the <strong>self</strong> system variable</td>
</tr>
<tr>
<td>output</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>io</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The following table lists the use of a parameter containing an object reference in a method.

<table>
<thead>
<tr>
<th>Usage</th>
<th>Reference Passed into Method</th>
<th>Object Referenced By Parameter Can Be Updated</th>
<th>Parameter Can Be Assigned New Object Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>input</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>output</td>
<td>No</td>
<td>Not applicable</td>
<td>Yes</td>
</tr>
<tr>
<td>io</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
When a parameter is an object reference, the usage of a parameter also determines the type of reference that can be passed by using the parameter, as listed in the following table.

<table>
<thead>
<tr>
<th>Usage</th>
<th>Types that Can Be Passed to Method</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>Reference of the same type as the parameter or any of its subclasses</td>
<td>Equivalent to the calling method assigning a value to the parameter in the method being called</td>
</tr>
<tr>
<td>input</td>
<td>Reference of the same type as the parameter or any of its subclasses</td>
<td>Equivalent to the calling method assigning a value to the parameter in the method being called</td>
</tr>
<tr>
<td>output</td>
<td>Reference of the same type as the parameter or any of its superclasses</td>
<td>Equivalent to the method being called assigning a value to the variable in the calling method</td>
</tr>
<tr>
<td>io</td>
<td>Reference passed must be of the same type as the parameter</td>
<td>Combination of input and output, so an exact match is required</td>
</tr>
</tbody>
</table>

For details about pseudo types and passing variable parameters to methods, see "Pseudo Types" and "Passing Variable Parameters to Methods", respectively, later in this chapter.

**Pseudo Types**

The PseudoType class is a special form of ‘indirect type’ that is used by the JADE compiler and has meaning only in the context of methods.

At compile time, a pseudo type is mapped to a single actual type or to a list of parameter types, depending on the Type in which the method is defined. (For details about parameter type lists, see "Passing Variable Parameters to Methods", later in this chapter.)

The pseudo types are as follows.

- InstanceType
- KeyType
- MemberType
- ParamListType
- SelfType
- PropertyListType

In all pseudo types, the compiler substitutes the pseudo type with a specific type when the method is compiled.

**InstanceType**

The InstanceType pseudo type is valid only in the context of methods defined for the Class class. At compile time, the InstanceType pseudo type maps to the Type of the instances of the class. For example, consider the signature of the Class::firstInstance method.

```java
firstInstance(): InstanceType is classFirstInstance in jomsupp;
```

When used in a statement similar to the following code fragment, the type of Customer.firstInstance expression matches the receiver class object, which in this case is the Customer class.

```java
firstCustomer := Customer.firstInstance;
```
KeyType

The KeyType pseudo type is valid only in the context of methods defined in the Dictionary class and subclasses. KeyType provides an indirection used by the compiler that expands to a list of keys defined in the receiver type.

The compiler expands the KeyType formal parameter, substituting an ordered list of parameters of the correct type for each of the keys in the dictionary. (See also the ParamListType pseudo type, which maps to a variable list of parameters of any type and is in effect a generalization of KeyType.) For example, the following is the signature of the Dictionary::getAtKey method.

\[ \text{getAtKey}(\text{keys}: \text{KeyType}) : \text{MemberType is dictGetAtKey} \text{ in jomsupp;} \]

The getAtKey method is inherited by any user-defined subclasses of the ExtKeyDictionary and MemberKeyDictionary classes. When the getAtKey method is used in a method, the compiler looks up the type of the receiver, which must be a dictionary to determine the dictionary keys. The compiler expands the KeyType formal parameter and substitutes an ordered list of parameters of the correct type for each of the keys in the dictionary.

MemberType

The MemberType pseudo type is valid only in the context of methods defined in the Collection class and subclasses. The MemberType maps to the member type of the Collection class in which the using method is defined. For example, the following is the signature of the Dictionary::getAtKey method.

\[ \text{getAtKey}(\text{keys}: \text{KeyType}) : \text{MemberType is dictGetAtKey} \text{ in jomsupp;} \]

See also "Passing Variable Parameters to Methods", later in this chapter.

ParamListType

The ParamListType pseudo type maps to a variable list of parameters of any type, and is in effect a generalization of the KeyType pseudo type. A method on any class can have a single ParamListType formal parameter, which must be the first or last formal parameter in the method signature.

When a method with a variable parameter list is called from a JADE method, the compiler allows a variable number of actual parameters of any type to be passed. For example, the following is the signature of the Object::sendMsgWithParams method.

\[ \text{sendMsgWithParams}(\text{msg}: \text{String}; \text{paramList}: \text{ParamListType}) : \text{Any is objSendMsgWithParams} \text{ in jomsupp;} \]

Note The ParamListType pseudo type can be used only as a formal parameter in a method signature. You cannot define a local variable with a type of ParamListType.

A compiler syntax error is raised when an attempt is made to pass parameters to a method that takes none, including a ParamListType parameter that may be empty.

For details about passing variable parameters to methods, see "Passing Variable Parameters to Methods", later in this chapter. For details about retrieving the value of a parameter in the parameter list at the specified position or returning the number of entries in a parameter list, see the Application class getParamListTypeEntry or getParamListTypeLength method, respectively, in Volume 1 of the JADE Encyclopaedia of Classes.

SelfType

The SelfType pseudo type maps to the type of the class in which the using method is defined. For example, the following is the signature of the Control::getControl method.

\[ \text{getControl}(\text{id}: \text{Integer}) : \text{SelfType is JadeControlGetControl} \text{ in jadpmap;} \]
The following example shows the signature of the `Object::cloneSelf` method.

```plaintext
cloneSelf(bTransient: Boolean): SelfType is objClone in jomsupp;
```

**PropertyListType**

The `PropertyListType` pseudo type maps to a list of properties that can be passed as parameters to a method. Use this pseudo type, for example, to specify a variable parameter list of one or more parameters of type `Property`.

A method on any class can have a single `PropertyListType` formal parameter, which must be the first or last formal parameter in the method signature.

When a method with a variable property list is called from a JADE method, the compiler allows a variable number of actual parameters of type `Property` to be passed.

```plaintext
addMemberKeyProperty(properties: PropertyListType);
```

**Note** The `PropertyListType` pseudo type can be used only as a formal parameter in a method signature. You cannot define a local variable with a type of `PropertyListType`.

For details about passing variable parameters to methods, see "Passing Variable Parameters to Methods", in the following subsection.

**Passing Variable Parameters to Methods**

JADE supports the passing of variable parameters to methods in specific situations, so that you can pass variable lists of dictionary-key parameters to methods. You can:

- Call dictionary methods where the keys are not known until run time; for example, use the key access methods provided defined in the `DynaDictionary` class to perform ad hoc queries or collection-based sorts without the overhead of maintaining multiple persistent dictionaries.
- Define your own generic methods of an abstract dictionary class.
- Define and call external methods with a variable list of parameters.

**Note** If the number or type of the actual parameters passed to a method by a parameter list does not correspond exactly to the formal parameter list declaration, an exception or an unpredictable result may occur, as the compiler is unable to perform any type checking on the values that are passed to a parameter list.

For details about variable parameter usage, see "Variable Parameter List Considerations", "Defining and Using External Dictionary Methods", and "Using Generic JADE Dictionary Methods", in the following subsections. See also "MemberType" and "ParamListType", earlier in this chapter.

**Variable Parameter List Considerations**

When using variable parameter lists, you should be aware of the following restrictions. (See also "Defining and Using External Dictionary Methods" and "Using Generic JADE Dictionary Methods", later in this chapter.)

**De-referencing Variable Parameters in a JADE Method**

You cannot de-reference or manipulate individual parameters received in a variable parameter list from within a JADE method.
Usage Restrictions

When using variable parameters in your JADE methods:

- Only one variable parameter list is permitted for each method. This parameter list must occur as the first or the last item defined in the formal parameters.
- A parameter list cannot be updated by the read instruction.
- The declared usage of the variable parameter list applies to all actual parameter values passed when the method is called. For example, if the usage of a variable parameter list is io or output, constants and literals may not be passed.

Variable Parameter Lists Cannot be Amended

You cannot add or remove parameters from a parameter list. When calling a method that has a variable parameter list, the actual parameter must be a list of simple parameters or a parameter list.

Variable Parameter Lists Cannot be Passed to External Functions

Variable parameter lists cannot be passed to external functions, nor can an external function be declared with a formal parameter that is a parameter list.

No External (C++) Representation of Variable Parameter Lists

When an external method is called, JADE expands any parameters in a variable parameter list, along with any fixed parameters, and constructs a single DskParamList parameter.

Defining and Using External Dictionary Methods

When a method with a KeyType formal parameter is an external method, the JADE compiler generates a fixed actual parameter list for the call. The transient DynaDictionary class encapsulates the behavior required to access entries in member key dictionary subclasses; that is, in dictionaries in which the keys are properties in the member objects. In addition, the DynaDictionary class enables you to defer the specification of the membership and keys until run time.

Dynamic dictionaries are useful in applications with requirements for:

- Ad hoc queries or collection-based sorts without the overhead of maintaining multiple persistent dictionaries
- Intensive collation or collection-based sorting

For examples, see "Using Dynamic Dictionaries", in Volume 1 of the JADE Encyclopaedia of Classes. See also "Using Generic JADE Dictionary Methods" and "Variable Parameter List Considerations", elsewhere in this chapter.

Using Generic JADE Dictionary Methods

JADE enables you to write generic dictionary methods that receive and pass on KeyType parameters in dictionary classes when no keys are defined. For example, JADE implements the Dictionary class includesKey method, which has a C++ implementation, as follows:

```pascal
includesKey(keys: KeyType): Boolean;
beg
  return getAtKey(keys) <> null;
end;
```

When a JADE method with a KeyType formal parameter is compiled in a dictionary class in which no keys are defined, a variable key list is assumed.
The following is an example of a method that could be defined at the Dictionary class level. It takes a KeyType parameter, passes it on to another dictionary method, and performs some additional computation.

```plaintext
def getAtKeyGeqOrLast(keys: KeyType): MemberType;
vars
    entry : MemberType;
begin
    entry := getAtKeyGeq(keys);
    if entry = null then
        // If not found, return the last entry in the dictionary
        entry := last;
    endif;
    return entry;
end;
```

Although this `getAtKeyGeqOrLast` method is a generic method defined in an abstract dictionary class that has no defined keys, it could be called using a reference to a dictionary type with defined keys. This enables the compiler to perform full type-checking on the keys.

In the following example, the `CustomerByNameDict` dictionary has a single string key. The actual parameter in the call in this example can be fully type-checked by the compiler.

```plaintext
vars
customerDict : CustomerByNameDict;
cust : Customer;
begin
    ...  
cust := customerDict.getAtKeyGeqOrLast("Wallace");
    ...
end;
```

See also "Defining and Using External Dictionary Methods" and "Variable Parameter List Considerations", earlier in this chapter.

**Return Type**

The `return-type` is the type of the result value returned by the method.

**Method Options**

You can specify the following method options.

- `abstract`
- `clientExecution`
- `condition`
- `conditionSafe`
- `final`
- `lockerReceiver`
- `mapping`
- `notImplemented`
- `partitionMethod`
- protected
- serverExecution
- subschemaCopyFinal
- subschemaFinal
- subschemaHidden
- typeMethod
- unitTest
- unitTestAfter
- unitTestAfterAll
- unitTestAfterClass
- unitTestBefore
- unitTestBeforeAll
- unitTestBeforeClass
- unitTestIgnore
- updating
- webService

For details, see the following subsections.

**Notes** If you specify any combination of two or more method options (for example, both the protected and updating method options in a method signature), they must be separated by a comma, to enable the method to compile. For example:

```java
myMethod() updating, protected;
```

For details about the presentationClientExecution and applicationServerExecution method options defined in the function definition of an external function when running in JADE thin client mode, see "Calling External Functions from JADE Thin Clients", in Chapter 1 of the JADE External Interface Developer's Reference.

---

**abstract Option**

The abstract method option indicates that the method is an abstract method.

An abstract method defines the method signature for a class, while the implementation of the method is deferred to the subclasses of the class. An abstract method has no source code associated with it and is never executed. Only abstract classes can have abstract methods.

**Note** Alternatively, you can specify this method option by using the Abstract check box in the JADE Method Definition or External Method Definition dialog. By default, this option is unset.

The following example shows the use of the abstract option in a method signature.

```java
drawData(window: Window input) abstract;
```
**clientExecution Option**

The `clientExecution` method option causes the method to be executed on the standard client or application server node. Methods that are called by the `clientExecution` method that do not specify the execution location continue to be executed on the standard client of application server node. When the `clientExecution` method completes, execution returns to the node or the browser from where the method was called.

If the method is called from a method executing on a server node, it enables windows on the client to be updated (for example, progress bars).

The method in the following example iterates through 5000 entries in the `Number` collection, and executes using the `clientExecution` method option.

```jade
executeOnClient() updating, clientExecution;
vars
    num : Number;
begin
    // When this method was tested on a presentation client in JADE
    // thin client mode, it took 4.277 seconds to execute, as the
    // method has to download the appropriate entry from the server
    // database at each iteration.
    beginTransaction;
    foreach num in numbers do
        num.key := 0;
    endfor each;
    commitTransaction;
end;
```

The method in the following example iterates through a collection of clients adding each client's name to an array, and executes using the `clientExecution` method option.

```jade
getAllOperators(): StringArray clientExecution;
vars
    client : Client;
    company : Company;
    result : StringArray;
begin
    company := Company.firstInstance;
    if company <> null then
        create result transient;
        foreach client in company.allClients do
            result.add(client.name);
        endfor each;
    endif;
    return result;
end;
```

See also "Server and Client Execution Restrictions" and the example under "serverExecution Option", later in this chapter.

**condition Option**

The `condition` method option indicates that the method is a condition, which is a declarative restricted method that returns a Boolean result. Conditions cannot be reimplemented from a superschema or superclass. You can define `constant` parameters for a condition that is not a constraint. Only `if` and `return` instructions can be used in condition methods.
Constraints are used in conditions to maintain automatic inverse references when the specified conditions are satisfied. Only automatic references can have a constraint. When the manual side of the inverse reference is set, the condition used as the constraint is evaluated and the automatic inverse is set only if the value of the condition is true. If the automatic reference is a Collection type, the condition is applied to the members of the collection.

For details about defining conditions, see "Adding Conditions to Classes or Primitive Types", in Chapter 4 of the JADE Development Environment User's Guide. See also "Parameters", earlier in this chapter.

The following example shows the use of the condition option in a method signature.

```plaintext
isMultipleOf(d : Integer): Boolean condition;
begin
  return 1 div d = 0;
end;
```

**conditionSafe Option**

The conditionSafe method option marks a mapping method as having no side effects.

When the mapping method of a virtual property is declared as conditionSafe, updates of the virtual property cannot save the new value, which makes the virtual property completely read-only.

Virtual properties with conditionSafe mapping methods can be used in RPS mappings. For details about specifying the display of virtual properties in the Relational Population Service wizard, see "Setting Up the RPS Options", in Chapter 15 of the JADE Development Environment User's Guide.

A mapping method marked with the conditionSafe method option has language limitations similar to those of a condition method. When using conditionSafe mapping methods, you:

- Can use only if and return instructions
- Can assign a value only to the second parameter
- Can use only the self system variable (for example, app and global are not allowed)
- Cannot declare and use local option variables
- Cannot use expressions with property de-references (for example, myCustomer.name is not allowed)
- Cannot reference an element in an array (for example, results[4] or the equivalent results.at(4))
- Cannot reference an object in a dictionary from its key value (for example, customers["Jones"] or the equivalent customers.getAtKey("Jones"))

**Note** The conditionSafe method option cannot be combined with the typeMethod method option.

**final Option**

The final method option indicates that the method cannot be reimplemented in a subclass. Use this option to restrict method reimplementation in this schema and its subschemas.

**lockReceiver Option**

The lockReceiver method option indicates that an exclusive lock of transaction duration is placed on the receiver object before the method is called. A check is also made that the process is in transaction state.
**Notes**  Methods defined on collections with the *updating* method option are regarded as also having the *lockReceiver* option, even if it is not declared explicitly.

The *lockReceiver* method option cannot be combined with the *typeMethod* method option.

**mapping Option**

The *mapping* method option indicates that the method is a mapping method. Mapping methods have the same name as a property and they are invoked automatically each time the value of the property is accessed in a method.

A mapping method has the following method signature.

```plaintext
method-name(set: Boolean; _value: type io);
```

Mapping methods act as a wrapper around properties. They can be used, for example, to ensure that properties are not updated with invalid values. If a corresponding property with the same name exists when a method is added, the method signature is then set up with the mapping option by default.

A virtual reference property must have a defined mapping method that is activated when the value of the property is set or retrieved. By default, methods are not mapping methods.

**Note**  If you have mapping logic on subclassed controls, other processes such as the JADE Painter, Translator utility, or the loading of schemas may also execute that logic. The logic therefore may need to perform checks to determine if it is running in the user application environment, to ensure that exceptions are not generated in these other situations.

When a mapped property is specified on the left of an assignment (for example, `prod.amount := amt`), the parameter value passed to the mapping method is *amt*. If the parameter value is changed to *newAmt* in the mapping method, the *amount* property becomes *newAmt*. When a mapped property is specified on the right of an assignment (for example, `amt := prod.amount`), the parameter value passed in the mapping method is *amount*. If the parameter value is changed to *newAmount* in the mapping method, the *amt* local variable becomes *newAmount*.

The *set* parameter is *true* if the property is being set and *false* if the value of the property is being read. The *variable* parameter is a variable value of the same type as the property.

**Notes**  Mapped properties (that is, properties that have a mapping method) from a *RootSchema* class cannot be used as dictionary keys.

If the mapping method for a property that is used as a key in a dictionary maintained by automatic inverse maintenance returns a value in the *variable* parameter that is not equal to the value of the property in the object, **1250 (Attempt to change key value in a mapping method get operation)** exception is raised.

The following example shows a mapping method for an *amount* property. The method automatically discounts any amounts over 500 by ten percent (10%) when they are set. When an amount is read, twelve and a half percent (12.5%) is automatically added.

```plaintext
amount(set: Boolean; value: Real io) mapping;
begin
  if set then
    if value > 500 then
      value := value * 0.9;
    endif;
  else
    value := value * 1.125;
```
end;
end;

For details about conditionSafe mapping methods, see "conditionSafe Option", earlier in this chapter, and for
details about creating a mapping method for a property selected in the Properties List of a hierarchy browser, see
"Adding a Method Mapping for a Property", in Chapter 4 of the JADE Development Environment User's Guide.

**Note** The mapping method option cannot be combined with the typeMethod method option.

**notImplemented Option**

The notImplemented method option indicates that an inherited method has no implementation defined in this
class and does not inherit an implementation (if one exists) from a superclass.

A method with the notImplemented option does not have a method body.

An attempt to call a method with the notImplemented option results in an exception at runtime. Compiling a
method that calls a method with the notImplemented option raises a syntax error.

**partitionMethod Option**

The partitionMethod method option specifies that the method is used to determine the partition in which the
receiver is to be stored. The method is used to specify the partition for an object in the following usage contexts:

- Automatically, when an object is created.
  
  The intent of a user-defined partition method is to determine the target partition for the receiver object, by
evaluating attributes of the receiver.
  
  The method in the following example reimplements the default autoPartitionIndex method of the Object
class that is invoked when an object is created

  ```
  autoPartitionIndex(): Integer partitionMethod;
  begin
    return completionDate.month();
  end;
  ```

- When a non-partitioned database file is converted into a partitioned one.
  
  A partition method other than a reimplementation of the autoPartitionIndex method can be defined and its
  name passed as an optional parameter to the JADE Database Administration utility (jdbadmin)
MakePartitioned action.

The following restrictions apply to a method marked with the partitionMethod method option.

- Can use only if and return instructions
- Can use only the self system variable (for example, app and global are not allowed)
- Cannot declare and use local variables
- Cannot use expressions with property de-references (for example, myCustomer.name is not allowed)
- Cannot use property expressions such as results[4] or the equivalent results.at(4), or customers["Jones"]
or the equivalent customers.getAtKey("Jones")
**protected Option**

The `protected` method option indicates that the method can be referenced only by methods in the same class or its subclasses.

**Note** Alternatively, you can specify this method option by using the **Protected** check box in the JADE Method Definition or External Method Definition dialog.

As the **protected** option is unset by default, you can use the methods access option on the **Schema** sheet of the Preferences dialog to change the default options for your schemas. (For details, see "Maintaining Schema Options", in Chapter 2 of the JADE Development Environment User's Guide.)

The following examples show methods with the **protected** option specified in the signature.

```plaintext
backupException(ex: Exception): Integer updating, protected;
begin
  if ex.errorCode = JErr_DbFileNotFound or
      ex.errorCode = JErr_DbUserAbort then
    return Ex_Resume_Next;
  endif;
  // pass all other exceptions to the next handler in the chain
  return Ex_Pass_Back;
end;

addressLine(line: Integer): String protected;
// returns a specific line of an address
vars
  addr1, addr2 : Integer;
  addressOut : String;
  char : Character;
begin
  addr1 := 0;
  addr2 := 1;
  while addr1 < address.length do
    addr1 := addr1 + 1;
    char := address[addr1:1].Character;
    if char <> Cr and char <> Lf and char <> ',' then
      addressOut := addressOut & char;
    else
      if addr2 = line then
        break;
      else
        addressOut := "";
        addr2 := addr2 + 1;
      endif;
    endif;
  endwhile;
  if addr2 = line then
    return addressOut.trimLeft;
  endif;
end;
```
serverExecution Option

The serverExecution method option indicates that the method and all methods subsequently called by this method are to be executed on the server node unless they are clientExecution methods, in which case they are executed on the node of the client calling the method. This method option provides performance benefits when a method accesses a large number of objects in multiuser mode, as the methods are executed on the node in which the objects reside instead of the required objects having to be passed to the client node for execution.

The method in the following example iterates through 5,000 entries in the Number collection and executes using the serverExecution method option.

```pascal
executeOnServer() updating, serverExecution;
vars
  num : Number;
begin
  // This method should take less time to execute than the same method
  // executing using the clientExecution option. When this method was
  // tested on a presentation client running in JADE thin client mode,
  // it took only 1.602 seconds to execute.
  beginTransaction;
  foreach num in numbers do
    num.key := 0;
  endforeach;
  commitTransaction;
end;
```

See also "Server and Client Execution Restrictions", later in this chapter, and the example under "clientExecution Option", earlier in this chapter.

Caution As the server node has its own private cache used by server execution methods, which is separate from the shared database cache, the edition of an object that is fetched from this private cache may not be the latest.

subschemaCopyFinal Option

The subschemaCopyFinal method option indicates that the method cannot be reimplemented in a subschema copy class. As an example, consider the following situation.

**Schema A**

Class C1 - method m1 - subschemaCopyFinal
Class C2
Class C3 - method m2 calls m1

As the developer of schema A, you may want to guarantee that when method m2 of class C3 is called, it always uses the schema A implementation of C1::m1. However, you will allow the C3 class to be subclassed and the m1 method to be reimplemented on subclasses. The following therefore applies.

**Schema B** - subschema of Schema A

Subschema Copy Class C1 (reimplementation of m1 not allowed on subschema copy)
Subschema Copy Class C2 (reimplementation of m1 not allowed because subschemaCopyFinal)
Subschema Copy Class C3 (reimplementation of m1 not allowed because subschemaCopyFinal)
Class C4 - reimplemented m1 allowed
subschemaFinal Option

The subschemaFinal method option indicates that the method can be extended or reimplemented in its local schema but not in a subschema.

Use the subschemaFinal option to make methods available to other schemas but prevent those schemas from modifying, reimplementing, or circumventing the defined method behavior. As an example, consider the following situation:

Schema A
Class C1 - method m1 : subschemaFinal
    Class C2 - subclass of C1
    Class C3 - subclass of C2

As the developer of schema A, you may want to specify that method m1 cannot be reimplemented in a subschema. Making method m1 in class C1 subschemaFinal accomplishes this.

subschemaHidden Option

The subschemaHidden method option indicates that the method is available only in the local schema; that is, it is not available for use in any subschemas.

The JADE compiler enforces that subschemaHidden methods are not referenced by subschema code, and subschema browser windows do not display methods that are subschemaHidden in a superschema.

typeMethod Option

The typeMethod method option indicates that the method is a type method.

A type method provides a way of calling a method declared on a type (class, primitive, or interface) without having to have an instance of the type. While the method is declared on a type and has the same scope as an instance method, at runtime the receiver is the type on which the method is declared; not an instance of the type.

Type methods enable you to define and call methods on class, primitive, or interface types without having to instantiate a dummy object, which simplifies aspects of JADE application development. Examples of a type method are:

- Factory method that creates and initializes instances of a class.
- Method that sums the values of all of the properties for all instances of a class (for example, a Product class could have a type method to compute the average price of all products).

A type method is identified by the typeMethod method option on the method declaration; for example:

demoTypeMethod() typeMethod;

Note Alternatively, you can specify this method option by using the Type Method check box in the JADE Method Definition or External Method Definition dialog. By default, this option is unset.

For details, see "Type Methods", later in this chapter.

unitTest Option

The unitTest method option indicates that the method is a unit test method.

The unit test method cannot have parameters or return a result.
The test fails if an assertion method fails or an exception is raised. Any failure associated with a method will count as a failure only in the final failure statistics, whether it is a failure resulting from any of the previous items in this list or from a failure in the `unitTest` method.

The `unitTest` method option in the following example identifies a unit test method that is executed.

```plaintext
add() unitTest;
begin
  calculator.add(1);
  calculator.add(1);
  assertEquals(calculator.getResult(), 2);
end;
```

**Note** The `unitTest` method option cannot be combined with the `typeMethod` method option.

### `unitTestAfter` Option

The `unitTestAfter` method option indicates that the method is a unit test method that is run after each unit test method of the class is executed.

Only one method in a `JadeTestCase` subclass can have the `unitTestAfter` option.

Any failure associated with a method will count as a failure only in the final failure statistics, whether it is a failure resulting from any of the previous items in this list or from a failure in the `unitTestAfter` method.

### `unitTestAfterAll` Option

The `unitTestAfterAll` method option indicates that the method is a unit test method that is run once after all classes have been executed.

**Note** Methods with `unitTestBeforeAll` and `unitTestAfterAll` method options can be defined only on the base `JadeTestCase` class, and there can be one instance only of these method options in a schema.

If any number of tests are run from the schema, the `unitTestBeforeAll` and `unitTestAfterAll` method options are executed once only.

If multiple `unitTestBeforeAll` and `unitTestAfterAll` method options are defined on methods of the `JadeTestCase` class in a schema, the first instance of the method option is used and a warning is logged, stating that all other instances have been ignored.

If `unitTestBeforeAll` and `unitTestAfterAll` method options are defined on methods in subclasses of the `JadeTestCase` class, they are ignored and a warning is logged, stating they have been ignored.

If a `unitTestAfterAll` method fails, the last class method executed is counted as failed.

A method with the `unitTestAfterAll` option specified can perform any one-off global clean up for all tests; for example, package finalization and deleting test data.

### `unitTestAfterClass` Option

The `unitTestAfterClass` method option indicates that the method is a unit test method that is run once after the class is tested.

Only one method in the class can have the `unitTestAfterClass` method option.

If a `unitTestAfterClass` method fails, the last class method executed will be counted as failed.
The `unitTestAfterClass` method option in the following example identifies the method that is run before running a group of unit test methods for a unit test class.

```javascript
finalize() unitTestAfterClass, updating;
begin
    app.finalize;
    // ... any other finalization
end;
```

**unitTestBefore Option**

The `unitTestBefore` method option indicates that the method is a unit test method that is run before each unit test method of the class is executed.

Only one method in a `JadeTestCase` subclass can have the `unitTestBefore` option.

Any failure associated with a method will count as a failure only in the final failure statistics, whether it is a failure resulting from any of the previous items in this list or from a failure in the `unitTestBefore` method.

The `unitTestBefore` method option in the following example identifies the method that is run before running each unit test method in a unit test class.

```javascript
clearTestBefore() unitTestBefore;
begin
    calculator.clear;
end;
```

**unitTestBeforeAll Option**

The `unitTestBeforeAll` method option indicates that the method is a unit test method that is run once before any classes are tested.

**Note**   Methods with `unitTestBeforeAll` and `unitTestAfterAll` method options can be defined only on the base `JadeTestCase` class, and there can be one instance only of these method options in a schema.

If any number of tests are run from the schema, the `unitTestBeforeAll` and `unitTestAfterAll` method options are executed once only.

If multiple `unitTestBeforeAll` and `unitTestAfterAll` method options are defined on methods of the `JadeTestCase` class in a schema, the first instance of the method option is used and a warning is logged, stating that all other instances have been ignored.

If `unitTestBeforeAll` and `unitTestAfterAll` method options are defined on methods in subclasses of the `JadeTestCase` class, they are ignored and a warning is logged, stating they have been ignored.

If a `unitTestBeforeAll` method fails, the first class method executed is counted as failed.

A method with the `unitTestBeforeAll` option specified can perform any one-off global initialization for all tests; for example, package initialization and creating test data.

**unitTestBeforeClass Option**

The `unitTestBeforeClass` method option indicates that the method is a unit test method that is run once before the class is tested.

Only one method in the class can have the `unitTestBeforeClass` option.

If a `unitTestBeforeClass` method fails, the first class method executed will be counted as failed.
The **unitTestBeforeClass** method option in the following examples identifies the method that is run before running a group of unit test methods for a unit test class.

```plaintext
turnonCalculator() unitTestBeforeClass, updating;
begin
    create calculator transient;
    calculator.switchOn;
end;

initialize() unitTestBeforeClass, updating;
begin
    app.initialize;
    // ... any other initialization
end;
```

**unitTestIgnore Option**

The **unitTestIgnore** method option indicates that the method is a unit test method that is ignored when the tests are run.

A possible reason for not running the test is that the unit test method is incomplete or the underlying functionality to be tested is incomplete.

The **unitTestIgnore** method option in the following example identifies a unit test method that is ignored when the tests are run.

```plaintext
multiply() unitTestIgnore; // not ready yet
begin
    calculator.add(10);
    calculator.multiply(10);
    assertEquals(calculator.getResult(), 100);
end;
```

**updating Option**

The **updating** method option indicates that the method can modify properties in the object to which it is sent; for example:

```plaintext
sysNotification(eventType: Integer;
theObject: Object;
eventTag: Integer) updating;
begin
    if eventType = Object_Create_Event or
    eventType = Object_Delete_Event then
        myForm.caption := "Customers " & allCustomers.size.String;
    endif;
end;
```

If you do not specify this method option, any instructions that update properties in the **self** object (that is, the object that is executing the method) or that attempt to call update methods for the **self** object will result in errors at compile time.

**Note** Alternatively, you can specify this method option by using the **Updating** check box in the JADE Method Definition or External Method Definition dialog. By default, this option is unset.
A temporary value is created if the return value of a primitive method is passed to an updating primitive method. On completion of the updating method, this temporary value is discarded. In the following example, a temporary string is used to return the value from the call to the String primitive type `trimBlanks` method. This value (`abc`) is then passed to the String primitive type `replaceChar` method, which updates the temporary string to `zbc`. The temporary string is discarded after the call to the `replaceChar` method, leaving the local variable `str` unchanged.

```haskell
vars
  str : String;
begin
  str := 'abc';
  str.trimBlanks.replaceChar('a', 'z');
  write str;
end;
```

**Notes**  
Methods defined on collections with the updating method option are regarded as also having the lockReceiver option, even if it is not declared explicitly.

The updating method option cannot be combined with the typeMethod method option.

### webService Option

The `webService` method option indicates that the JADE method specifies a Web services method; that is, one that is defined in a subclass of the JadeWebService class or one of its subclasses. (For details, see "Creating Web Service Methods", in Chapter 11.)

You can specify any type of parameter (that is, `constant`, `input`, `io`, or `output`) for a Web services method. For details, see "Parameters", earlier in this chapter.

The method does not require a return value because the Web service uses the HTTP protocol (which always requires a response) and a reply is always sent to the Web service consumer. Primitive parameters that are `io` or `output` are also sent back in the response as are object parameters that are not `constant`.

**Notes**  
You can also specify this method by using the Web Service check box in the JADE Method Definition or Condition Definition dialog. For a Web services class, this option is set by default.

Methods declared on the JadeWebService class and its subclasses that are marked as Web service methods cannot have a return type of `Any` and cannot have parameters of type `Any`.

The following example shows the use of the `webService` option in a method signature.

```haskell
getClients() webService;
```

### Server and Client Execution Restrictions

The following restrictions apply when using the serverExecution or clientExecution method option.

- You cannot specify these method options on primitive type methods (for example, a `String` or `Integer` method). Methods defined on primitive types are always executed in the node of the calling method.

- Persistent transactions must be total client transactions or total server transactions; that is, any `beginTransaction` and `commitTransaction` pair of instructions must be done while executing on the client (without executing an updating server method), or while executing on the server (without the execution of updating client methods). This restriction applies also to the `beginLoad` and `endLoad` and the `beginLock` and `endLock` instruction pairs. Transient transactions (for shared transient objects) do not have this restriction.

- Persistent transactions must be started, performed, and finalized on a single node (that is, a client node or a
server node). All of the update operations of the transaction must occur in the same node that started the transaction. Applications should not attempt to access persistent objects that have been updated but not yet committed on the other node. The updates are usually not visible on the other node. A 1276 exception (Potential cache inconsistency) may be raised if this is attempted.

- You cannot debug server execution methods nor can a server method invoke a graphical user interface (GUI) method.
- You can access all GUI properties and methods (which are marked as clientExecution methods) from a server method except for anything that brings up a modal-type dialog (that is, the common dialog class methods and the Application::msgBox, Form::showModal, and Form::popupMenu methods). The other exceptions to this are the Application class doWindowEvents, checkPictureFile, and loadPicture methods, which are executed relative to the server.

**Caution** Use of GUI methods and properties is very expensive in a server method. A clientExecution method requires that all transient objects passed to the server are passed back with the client execution (and passed back to the server after the client execution is complete). Accessing GUI properties and methods within a server execution therefore should be done only with careful consideration.

A clientExecution method requires that all transient objects updated by the server method are passed back with the client execution (except for created non-shared transient objects). After the client execution is complete, transient objects accessed by the server method have to be fetched from the client node again.

### Constant Declaration

A constant-declaration has either of the following syntaxes.

```plaintext
name = constant-expression;

name : primitive-type = constant-expression;
```

The `name` must start with an uppercase letter and it cannot be the same name as a global constant. The `constant-expression` is an expression that contains only literal values or other constants.

The following example illustrates constant declarations.

```plaintext
constants
    SecondsInMinute = 60;
    SecondsInHour = SecondsInMinute * 60;
    SecondsInMinute = SecondsInHour * 24;
```

Constants declared in this way can be used in the body of the method, providing a more meaningful representation than simply using literal values.

For details about promoting a local constant definition to a type constant and promoting a method value to be a local method constant, see "Promoting a Local Constant to be a Type Constant" and "Promoting a Method Value to be a Local Method Constant", respectively, in Chapter 4 of the JADE Development Environment User's Guide.

### Variable Declaration

A variable-declaration has the following syntax.

```plaintext
name[, name, name...] : type;
```

The `name` must start with a lowercase letter.
The following example illustrates variable declarations.

```plaintext
vars
  count : Integer;
  name : String;
cust1, cust2 : Customer;
```

Variables used in this way can then be used as working storage for values in the body of the method.

For details about promoting a local variable to be a property on the class and introducing an unknown local identifier as a local variable when editing a JADE method, see "Promoting a Local Variable to be a Property on the Current Class" and "Introducing an Unknown Local Identifier as a Local Variable", respectively, in Chapter 4 of the JADE Development Environment User's Guide.

**JADE Instructions**

JADE instructions (or statements) form the body of your method and define the actions to be carried out as a sequence of instructions.

For more details, see "JADE Instructions, Expressions, and Assignments", later in this chapter.

**Epilog**

The epilog is an optional unit of code that is guaranteed to be executed before the method is exited, regardless of whether the method terminates normally or is terminated abnormally as a result of an exception.

For more details, see "Epilogs", later in this chapter.

**Using Comments in Your Methods**

You can include descriptive or documentary comments in your JADE methods.

The types of comments that can be used in a method are as follows.

- Comments beginning with double stroke character strings (//).
  
  This type of comment continues only until the end of the line on which it appears.

- Comments enclosed with stroke asterisk (*) and asterisk stroke (*!) character strings.
  
  This type of comment can span multiple lines.

**Note** In multiple-line comments, all text (including any double stroke (//) character string) that follows the stroke asterisk (*) character string is ignored until the asterisk stroke (*!) character string is encountered.

The following example shows the use of single-line comments and embedded comments.

```plaintext
helloWorld();
vars
  count : Integer;
begin
  // outputs "hello world" 10 times
  foreach count /* embedded comment */ in 1 to 10 do
    /* multiple
    line
    comment */
    write "hello world";
```
You can fold multiple-line block comments (that is, comments bounded by the \* and */ notation as well as multiple consecutive line comments that start with double stroke character strings (//)) in the editor pane, by selecting the Normal or Compact folding option on the Editor Options sheet of the Preferences dialog. By default, folding is not enabled. (For details, see "Maintaining Editor Options", in Chapter 2 of the JADE Development Environment User's Guide.)

When folding in the editor pane is enabled, you can expand and collapse multiple-line block comments. A minus icon (-) is displayed next to the first comment line that can be folded up to remove the majority of the comment from view. JADE enables you to fold multiple-block comments as follows.

- A stroke asterisk (*) and asterisk stroke (*\) comment block that stretches over more than one line, with the exception being a comment line started on the end of another line that can be folded for other reasons; for example:

  if ... then /*

- Multiple consecutive lines that start with double stroke character strings (//) comments where the // token is preceded only by spaces or tabs. If your folding option preference is set to:
  - Normal, the folding icon is displayed if three consecutive lines are in the format white space// ...
    (because folding the comments will show the first line underlined followed by the last comment line).
  - Compact, the folding icon is displayed if two consecutive lines are in the format white space// ...
    (because folding the comments will show only the first line underlined and not the last line).

Type Methods

Type methods provide a way of calling a method declared on a type (class, primitive, or interface) without having to have an instance of the type. While the method is declared on a type and has the same scope as an instance method, at run time the receiver is the type on which the method is declared; not an instance of the type.

Type methods enable you to define and call methods on class, primitive, or interface types without having to instantiate a dummy object, which simplifies aspects of JADE application development. Examples of a type method are:

- Factory method that creates and initializes instances of a class.
- Method that sums the values of all of the properties for all instances of a class (for example, a Product class could have a type method to compute the average price of all products).

A type method is identified by the typeMethod method option on the method declaration; for example:

```java
demoTypeMethod() typeMethod;
vars
begin
end;
```

**Note** As the receiver is not an instance of the type, you cannot use the self system variable in a type method, but you can use the selfType system variable, which references the type. You can use the selfType system variable in both type methods and existing class instance methods.

Two syntaxes are supported for calling a type method. To avoid overloading the dot operator (.) notation, the @ operator identifies that a type method is being called.
The following syntax uses the name of the type to call a type method using the name of the type.

The following example is a call to the `demoTypeMethod` type method in class `C1`.

```java
C1.demoTypeMethod();
```

In this example, the `@` operator notation makes it clear that the method being called is a type method defined on the type `C1`; not on the `Class` class, as would be the case with the `.` operator.

The following syntax calls a type method using a variable, parameter, or property.

The following example is a call to the `demoTypeMethod` type method on class `C1`, using a local variable of type `C1`.

```java
vars
    c1 : C1;
begin
    c1 := C1.firstInstance();
    c1.demoTypeMethod();
end;
```

In this example, the method can be called even if the value of `c1` is null. The variable is used by the compiler to determine the method to call; in this case the `demoTypeMethod` type method declared on the `C1` class.

**Note**  Because `demoTypeMethod` is a type method, the receiver is not the value of `c1` but the type of the value of `c1`.

See also "Invoking a Type Method", in the following subsection.

The Jade Method Definition dialog or the External Method Definition dialog contains the `Type Method` check box, which is unchecked by default. You can declare an existing method as a type method, by adding the `typeMethod` method option to the method signature and then recompiling the method. (Conversely, convert a type method to a class (instance) method, by removing the `typeMethod` modifier, removing any `selfType` system variable or changing the `selfType` system variable to `self`, and then recompiling the method.)

The Methods List of browser windows has three folders: All, Instance, and Type. By default, the All folder is displayed; that is, all instance methods and all type methods are displayed. By default, instance methods are displayed in a black font and type methods are displayed in a dark blue font in the Methods List of browser windows.

You can reimplement type methods on subclasses. As is the case for instance methods, the signature must be the same in all implementations. At run time, the value of the variable is used to determine which implementation of `demoTypeMethod` will be called. If the value of the variable is an instance of a subclass and a type method reimplements a type method on the class, the reimplemented type method will be called. If the value of the variable is null, the type method on the declared type of the variable is called.

Transient type methods (created by the `Process` class `createTransientMethod` method) are supported.

JADE implements the following Application Programming Interfaces (APIs) to call a type method on a class and primitive type. The `jomSendTypeMsg` API call sends a message to the type of the receiver, and the `jomCallPrimTypeMethod` API call invokes a primitive type; for example, from an external method. For more details, see "JADE Application Programming Interface (API)", in Chapter 3 of the JADE Object Manager Guide.

<table>
<thead>
<tr>
<th>A type method can...</th>
<th>A type method cannot...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Be a JADE method or an external method.</td>
<td>Use <code>self</code> or any instance methods or properties.</td>
</tr>
</tbody>
</table>
A type method can...

- Be defined on a JADE class, a primitive type (for example, `Integer`), or an interface type.
- Be used with the `public`, `updating`, `protected`, `serverExecution`, and `clientExecution` method options.
- Reimplement a type method with the same signature defined on a superclass but it cannot reimplement an instance method (and the reverse).
- Invoke its superclass implementation using the JADE `InheritMethod` instruction or the C++ `jomInheritMethod` API.
- Be declared on a subschema copy class, with the usual JADE method visibility applying (that is, a subschema copy method cannot be accessed by applications running in a superschema or a peer schema if the root type or a subschema copy for the class already exists in the schema or a superschema).
- Be imported from a package.
- Be defined on an imported class.

A type method cannot...

- Be a constructor or a destructor.
- Be a notification or an event method.
- Be a condition.
- Be combined with the `lockReceiver`, `updating`, `mapping`, or `webService` method option.
- Reimplement a method defined on a subschema copy class in a superschema or on the root class if the type method is defined on a subschema copy class.
- Be `conditionSafe`.
- Be used in a `unitTest` method.
- Be called from a REST service.

### Invoking a Type Method

You can call a type method by qualifying the method call with the name of the class to type, as shown in the following code fragment, which specifies the `someMethod` method on the `Company` class.

```java
Company@someMethod();
```

Call a type method exported from a package in the same way, as shown in the following code fragments.

```java
ImportedCompany@someMethod();
Package::Company@someMethod();
```

You can provide an application context for the call to a type method, as shown in the following code fragment.

```java
Company@someMethod() in someApplicationContext;
```

You can also call a type method by using a variable, parameter, or property. In this case, the type of the value of the variable identifies the type method to call. With the exception of the JADE `InheritMethod` instruction, every call to a type method must be qualified. This includes a method calling a type method on the class of the receiver (or a superclass of the receiver), and a type method calling another type method. Even though the methods are in scope, they must be qualified to make it clear that a type method is being called.

You cannot call a type method indirectly using the `Object` class `sendMsg` or `sendMsgWithParams` method, nor can you explicitly call an exported type method using the `Object` class `invokeMethod` method. Illegal calls to a type method result in the error 1185 (*Invalid call to a type method*).

Calling an instance method with one of the `Object` class methods `sendTypeMsg`, `sendTypeMsgWithParams`, or `sendTypeMsgWithIOParams` results in the error 1184 (*The requested method is not a type method*).
Accessing Meta Data

Metaschema objects define the structure of your schemas, classes, interfaces, and so on. The :: meta-object scope operator enables you to access metaschema objects by using a simple syntax to identify the object to be referenced. For details, see the following subsections.

- Accessing a Property, Constant, or Method in a Metaschema
- Meta Data Access Restriction
- Accessing a Meta-Data Class in an Imported Package

Accessing a Property, Constant, or Method in a Metaschema

Use the :: meta-object scope operator to reference a property, constant, or method in a type, in the following format.

\[ \text{type-name}::\text{property-or-constant-or-method-name} \]

The code fragments in the following example show the use of the scope operator to access a property, a constant, and a method.

```plaintext
write Product::code.name; // Outputs "code"
write Product::Obsolete.name; // Outputs "Obsolete"
write Customer::update.name; // Outputs "update"
```

The :: meta-object scope operator can be useful when specifying a property or method name as a parameter to a method. For example, you could use the code fragment shown in the following example when calling the `TcpIpConnection` class `listenContinuousAsynch` method passing the class C1 method `tcpCallBackMethod`.

```plaintext
listenContinuousAsynch(someC1Obj, C1::tcpCallBackMethod.name);
```

The advantage of using this notation is that should you rename the method, it automatically renames and recompiles any references to this method that are constructed in this manner.

Meta Data Access Restriction

Metaschema objects used in a method are identified at compile time. Consequently, you cannot identify a metaschema object by using an object reference, as shown in the following example.

```plaintext
vars
cust : Customer;
begin
cust := Customer.firstInstance;
write cust::address.length; // Error, as invalid syntax
write Customer::address.length; // Correct
end;
```

The following usages are allowed.

- `ClassName::method` // if the method is valid for browserExecution
- `ClassName::property` // if the property is valid for browserExecution

The following usages are not allowed.

- `ClassName::Constant`
- `ClassName::property.xxx` // where ".xxx" attempts to de-reference
ClassName::method.xxx  // the property or method further

**Accessing a Meta-Data Class in an Imported Package**

You can explicitly reference a class object in an imported package by qualifying the type name (that is, a class or interface) with its package name, using the following syntax.

```
imported-package-name::type-name
```

The :: meta-object scope operator specifies the explicitly qualified name of the metaschema object, as shown in the following example.

```
vars
cust : CustomerPkg::Customer;
begin
  cust := CustomerPkg::Customer.firstInstance;
  ...
  // Do some processing here
end;
```

**Tip**  When an imported type can unambiguously be identified, you do not have to prefix the type name with the imported package alias. When an imported type name does conflict with a local or inherited type, the unqualified name always refers to the local or inherited type. To access the imported class in such cases, prefix the imported class name with the imported package name. For details about packages, see Chapter 8, "Using Packages".

---

**JADE Instructions, Expressions, and Assignments**

This section describes JADE:

- Instructions
- Expressions
- Assignments

For details, see the following subsections.

**Instructions**

The body of a JADE method defines the actions to be carried out as a sequence of instructions, where each *instruction* specifies one corresponding action. JADE is a sequential programming language; that is, instructions are executed sequentially, one after the other, and not simultaneously.

JADE provides a variety of instruction forms, both simple and structured, to enable you to specify method actions.

**Expressions**

An *expression* is a rule for computing a value.

An expression consists of one or more operands combined by means of language-defined operators. Operands used in expressions can be one of the following.

- A literal value; for example, 1, true, null, "hello"  (for more details, see "Literals", later in this section)
- A constant value; for example, Red, SecondsInDay  (for more details, see "Constants", later in this section)
- A property (attribute or reference)
Literals

Literals in expressions are literal values that can be:

- A number.
- A string (can be enclosed in double (""") or single (") quotation marks).
- A character (any one-character string).
- The JADE-defined (that is, system-defined) Boolean literals:
  - true
  - false

Use these true and false system-defined Boolean literal values to initialize a Boolean variable.

- The JADE-defined null literal, which represents the null, or uninitialized, state of primitive type values or object references. Use the system-defined null literal to initialize an object reference or compare for an empty object, as shown in the code fragments in the following examples.

  ```java
  if customer <> null then
      customer.name := "John Doe";
  endif;
  
  if customer = null then
      write "Uninitialized customer";
  else
      write customer.name;
  endif;
  ```

See also "StringUtf8 Literals", "Type of a Numeric Literal", and "Hexadecimal Literals", in the following subsections.

StringUtf8 Literals

A StringUtf8 literal is enclosed in double ("") or single (") quotation marks, and is usually preceded by an at sign (@), as shown in the following example.

  ```java
  stringUtf8 := @"Jade Software";
  ```

If all of the characters are ASCII characters, as in the preceding example, the at sign is optional. The StringUtf8 literal can contain a non-ASCII character, by enclosing a value representing the character between an ampersand character (&) and a semicolon character (;), as shown in the following examples.

  ```java
  stringUtf8 := @"Copyright &copy; Jade Software";
  ```
In the first example, a character entity reference defined in the HTML 4 standard is used.

In the second and third examples, the value of the Unicode code point of the character in decimal and in hexadecimal is used. To retrieve a list of supported character entity names and values, use the dumpCharacterEntityTable method of the System class.

**Types of Numeric Literals**

The JADE compiler treats a numeric literal as an Integer, Integer64, Decimal, or Real primitive type, according to the following rules.

If the literal does not contain a decimal point, it is treated as an Integer, Integer64, or Decimal primitive type, depending on the value. The following table lists the ranges for positive integer values.

<table>
<thead>
<tr>
<th>Range</th>
<th>Implicit Type of Literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 through 2,147,483,647 (Max_Integer)</td>
<td>Integer</td>
</tr>
<tr>
<td>2,147,483,648 through 9,223,372,036,854,775,807 (Max_Integer64)</td>
<td>Integer64</td>
</tr>
<tr>
<td>Greater than 9,223,372,036,854,775,807 but less than 1024</td>
<td>Decimal</td>
</tr>
</tbody>
</table>

The ranges for negative integer values mirror those for positive integer values, as listed in the following table.

<table>
<thead>
<tr>
<th>Range</th>
<th>Implicit Type of Literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>-2,147,483,647 through 0</td>
<td>Integer</td>
</tr>
<tr>
<td>-9,223,372,036,854,775,807 through -2,147,483,648</td>
<td>Integer64</td>
</tr>
<tr>
<td>Less than -9,223,372,036,854,775,807 but greater than -1024</td>
<td>Decimal</td>
</tr>
</tbody>
</table>

If the literal contains a decimal point, it is treated as a Decimal or Real primitive type, depending on the number of significant digits, as listed in the following table.

<table>
<thead>
<tr>
<th>Number of Significant Digits</th>
<th>Implicit Type of Literal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 through 15</td>
<td>Real</td>
</tr>
<tr>
<td>Greater than 15</td>
<td>Decimal</td>
</tr>
</tbody>
</table>

**Hexadecimal Literals**

A number sign (#) precedes a hexadecimal (hex) value. You can define literal hexadecimal integer, string, character, or binary values.

Hexadecimal strings in JADE are represented as a sequence of space-separated hexadecimal characters.

The following example illustrates the use of string and integer hexadecimal literals in defining a series of constants.

```java
constants
    WhiteSpace = 
                   "09 0A 0B 0C 0D 20";
    MaxInt = #FFFFFFFF;
    MaxInt16 = #FFFF;
    BitMask01 = #55555555;
```
To define a binary hexadecimal literal value, enclose a sequence of space-separated hexadecimal characters in bracket symbols ([ ]). For example:

```plaintext
DaysInMonth = #1F 1C 1F 1E 1F 1E 1F 1E 1F 1F 1E 1F 1E 1F 1E 1F;
```

The following `daysInMonth` method of the `Integer` primitive type provides an example of the use of a binary hexadecimal literal to return the number of days in the specified month. This method expects a Boolean parameter advising if the year in question is a leap year.

```plaintext
daysInMonth(isLeapYear: Boolean): Integer;
constants
  // construct a compile-time value array
  DaysInMonth = #1F 1C 1F 1E 1F 1E 1F 1E 1F 1E 1F 1E 1F 1E 1F 1E;
begin
  if self = 2 and isLeapYear then
    return 29;
  endif;
  return DaysInMonth[self];
end;
```

**Constants**

`Constants` in expressions can be user-defined constants or one of the predefined system constants.

**Notes** Global constants are globally visible in a schema. However, as class and interface constants apply only to the class or interface in which they are defined (that is, more than one class or interface can have a constant of the same name), you must specify the class or interface to which a constant applies if referencing it outside of the defining class; for example:

```plaintext
myTbl.alignment := Table.Alignment_Center;
```

To access a constant defined in an interface, qualify it as `interface-name.constant-name`, not `class-name.constant-name`.

**System Variables**

The system variable operands that can be used in the JADE language are:

- app
- appContext
- currentSchema
- currentSession
- exception
- global
- method
- node
- process
- rootSchema
- self
selfType

system

For details, see the following subsections.

**Note**  Lock environmental object collections (for example, `node processes` and `system.nodes`) with caution, as this can cause hold-ups when processes sign off and on and when nodes initiate and terminate.

### app

The `app` system variable references the current transient application instance. JADE automatically creates a unique `app` object for each JADE application that is running and for each package that is imported (recursively) by the schema from which that application is running.

You can use the `app` object to store data that is global to the running application or as a place to implement application-specific code.

The `app` object is a transient instance of the `Application` subclass for the current schema. This class is created automatically by JADE when a schema is created and it inherits from the `Application` subclasses of all superschemas.

You can define additional properties and methods in your own application class, if required. For example, the following code fragments access the transient `app` instance.

```plaintext
write "The current time is " & app.actualTime.String;

if customer.name = "" then
    app.msgBox("Please enter a name", "Error", MsgBox_OKOnly);
endif;
```

**Note**  If a method is executing on the server node, the `app` instance may not be resident, resulting in a fetch from the client node (whereas if you send a message to `self`, the current receiver of the method that is executing is guaranteed to be resident in cache).

### appContext

The `appContext` system variable references the current transient application context instance. JADE automatically creates a unique `appContext` object for each JADE application that is running and for each package that is imported (recursively) by the schema from which that application is running. The `appContext` is a transient instance of the `ApplicationContext` class.

**Note**  The `appContext` system variable is of particular use when using packages, as it enables a package to switch context back to the context of the application that imported the package, using the `Object` class `invokeMethod` method. (For details about packages, see Chapter 8, "Using Packages").

### currentSchema

The `currentSchema` system variable references the current schema instance. The following example outputs the names of all locales supported by the current schema.

```plaintext
vars
    objArray : ObjectArray;
    obj : Object;
begin
    create objArray transient;
```
currentSchema.getAllLocales(objArray);  
foreach obj in objArray do  
  write obj.Locale.makeLocaleName;  
endforeach;  
epilog  
  delete objArray;  
end;

currentSession

The `currentSession` system variable references the current Internet session, to enable you to return the current session when you have created a subclass of the `WebSession` class.

The code fragment in the following example shows the use of the `currentSession` system variable.

```
// Show Exit and File menu options only if not running on the Web
if currentSession <> null then
  menuFile.visible := false;
  miExit.visible  := false;
endif;
```

**Note** The `currentSession` system variable is set to null if there is no current Web session.

exception

For details about the use of the `exception` system variable, see "on `exception` Instruction", later in this chapter.

The code fragment in the following example shows the use of the `exception` system variable.

```
on FileException do fileExceptionHandler(exception);
```

global

The `global` system variable references a global persistent object that is automatically created by JADE. There is a single `global` object for each schema, which is shared by all applications running from that schema.

You can use the `global` object as a means of exchanging information between applications or to retain information in a persistent form beyond the lifetime of a specific application.

The `global` object is an instance of the `Global` subclass for the current schema. This class is created automatically by JADE when a schema is created and it inherits from the `Global` subclasses of all superschemas. For example, consider an application that must record its most recent start-up time on the global object. The following attribute has been added to the `Global` subclass.

```
startupTime : TimeStamp;
```

Additionally, the `initialize` event method in the following example has been implemented.

```
initialize() updating;
vars
begin
  beginTransaction;
  global.startupTime := app.actualTime;
  commitTransaction;
end;
```
method

The method system variable references the currently executing method. For example, you could use the code fragment in the following example to trace the name of any method.

```
write method.name;  // outputs method name to the display window
```

node

The node system variable references the current instance of the Node class (the workstation that hosts the execution of the current process).

One node object exists for each logical workstation connected to the server node workstation. There is one fixed server node and none or many client nodes. A node represents a workstation that hosts the execution of one or several processes and it contains a dictionary of the processes currently active in the node.

A node object is created for each JADE executable program that is running; that is, a workstation that is running two JADE applications has two node objects, or logical workstation connections to the server.

The following example writes out the user codes of all processes running on the current node.

```
vars
  proc : Process;
begin
  write "Processes running on " & node.name;
  foreach proc in node.processes do
    write proc.userCode;
  end;
end;
```

process

The process system variable references the current thread in a workstation executing the current method. The code fragment in the following example writes the usercode and sign-on time of the current process.

```
write process.userCode & ": sign on time = " & process.signOnTime.String;
```

The code fragment in the following example puts the current process to sleep for five seconds.

```
process.sleep(5000);
```

rootSchema

The rootSchema system variable references the RootSchema, as shown in the code fragment in the following example.

```
write rootSchema.name;  // outputs "RootSchema" to the display window
```

self

The self system variable references the current instance (method receiver), and can be used in any expression. You can use the self system variable on the left of an assignment only in an updating primitive method.

When referring to a property or method of the current instance, you do not have to reference the self system variable. For example, to access a pea property in the receiver object of a method, you do not have to write self.pea, as simply writing pea is sufficient.
However, there is sometimes a requirement to refer to self explicitly. The following is an example of a method of the String primitive type, which replaces all occurrences of a character with another character.

```java
replaceCharacter(char: Character; withChar: Character) updating;
vars
count : Integer;
begin
  foreach count in 1 to self.length do
    if self[count] = char then
      self[count] := withChar;
    endif;
  endforeach;
end;
```

### selfType

The selfType system variable references the type of the current method receiver. You can use the selfType system variable in both type methods and instance methods.

### system

The system system variable references the instance of the System class (the group of nodes to which the current node belongs). For example, you could use the code in the following example to access the locks in all nodes in the JADE environment.

```java
vars
  locks : LockArray;
  notMine : Lock;
begin
  create locks transient;
  system.getLocks(locks, 100); // retrieve a maximum of 100 locks
  foreach notMine in locks do
    write notMine.target.String & "locked by " & notMine.lockedBy.userCode;
  endforeach;
epilog
  delete locks;
end;
```

### Expressions with Operators

JADE provides the following operators to enable you to construct composite expressions.

- **Unary**
- **Binary**
- **Substring**

For details, see the following subsections. See also "Operator Precedence", later in this chapter.
Unary Operators

The plus symbol (+) and the minus symbol (-) are unary operators applicable to numerical expressions. The - operator has the effect of negating the expression that follows it.

The + operator has no effect, and is included in the language for symmetry purposes only. These + and - unary operators are not applicable to Boolean expressions.

The not unary operator is applicable to Boolean expressions. It has the effect of logically negating the Boolean expression that follows it.

Binary Operators

Binary operators, which take exactly two operands, are:

- Arithmetic Binary Operators
- Boolean Binary Operators
- Relational Binary Operators
- String Concatenation Operator

For details, see the following subsections.

Arithmetic Binary Operators

Numerical operands can be combined, by using the binary operators listed in the following table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Real division (for example, 7 / 3 = 2.33333333333333)</td>
</tr>
<tr>
<td>div</td>
<td>Integer division (division with truncation; for example, 7 div 3 = 2)</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation (for example, x^3 is x cubed)</td>
</tr>
<tr>
<td>mod</td>
<td>Modulus (remainder after integer division)</td>
</tr>
</tbody>
</table>

The code fragments in the following examples show the use of arithmetic binary operators.

```plaintext
if (shift div 2) mod 2 = 1 then  // test if Ctrl key is pressed
    while result > 0 do
        remainder := result mod 16;
        result := result div 16;
        outString := hexVals[remainder + 1] & outString;
    endwhile;
```

Notes

The operands of the div operator can be any mix of Integer, Integer64, Real, or Decimal. Real and Decimal operands are typecast to an Integer before the div operation is performed, and must therefore be in the range supported by the Integer type. A divide by zero exception occurs when a divisor between zero (0) and one (1) is typecast and truncated.

If either operand of a div operation is of type Integer64, the result is also of type Integer64.
You can make arithmetic operations between numeric expressions that have a different type, as shown in the following example.

```jade
vars
    int : Integer;
    dec : Decimal[4,2];
begin
    int := 7;
    dec := 5.2;
    write int + dec;
end;
```

Before the operation is carried out, the expression with the lower numeric type is converted to the type of the other expression. In the following list of the order of numeric types, the Byte primitive type has the lowest order and Decimal the highest.

1. **Byte**
2. **Integer**
3. **Integer64**
4. **Real**
5. **Decimal**

In the previous example, the `int` operand is converted to Decimal (the same type as the `dec` operand) before the comparison is made.

**Boolean Binary Operators**

JADE provides standard operators that take Boolean values as operands and produce a Boolean result. These operators include the logical:

- **and**
- **not** (negation)
- **or** (inclusive)

Applying relational operators to operands of other types can also produce Boolean values. The following table lists the action and resulting values of the **and**, **or**, and **not** operators.

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>p and q</th>
<th>p or q</th>
<th>not p</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

Boolean expressions involving the **and** or the **or** operators are conditionally evaluated; that is, in the expression `a and b`, `b` is not evaluated if `a` is false.

In the expression `a or b`, `b` is not evaluated if `a` is true. This may be important in some cases where `a` and `b` are methods with side effects.
and Operator

The syntax of the and operator is:

boolean-expression and boolean-expression

The code fragment in the following example shows the use of the and operator.

```java
if newManager = null and newCompany.allEmployees.size > 0 then
  app.msgBox("A manager is required", "Error", MsgBox_OK_Only);
  return;
endif;
```

not Operator

The syntax of the not operator is:

not boolean-expression

The code fragment in the following example shows the use of the not operator.

```java
if not file.isOpen then
  app.msgBox("File is not open", "Error", MsgBox_OK_Only);
  return;
endif;
```

or Operator

The syntax of the or operator is:

boolean-expression or boolean-expression

The code fragment in the following example shows the use of the or operator.

```java
if newLevel < 1 or newLevel > 5 then
  app.msgBox("Invalid level", "Error", MsgBox_OK_Only);
  return;
endif;
```

Relational Binary Operators

A relational expression consists of two operands separated by a relational operator. If the relation is satisfied, it has the value true. If the relation is not satisfied, it has the value false.

For details about using relational binary operators, see "Using Relational Binary Operators" and "Comparing Numerical Entities of Different Types", in the following subsections.

The result of a relational operation is therefore a Boolean value, as shown in the following examples.

```java
isMarried(): Boolean;
begin
  return spouse <> null;
end;

isAParent(): Boolean;
begin
  return allChildren.size >0;
end;
```
The relational binary operators are listed in the following table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
</tbody>
</table>

Using Relational Binary Operators

Use the = and <> operators to compare any compatible pair of object references or primitive values. Two object references are considered equal if they refer to the same object. If they refer to different objects, they are considered unequal, even if the objects have identical property values.

Use the <, >, <=, and >= operators to compare primitive values (or expressions that produce primitive values) only. These operators have the following meaning.

- For numeric operands (Byte, Integer, Integer64, Real, or Decimal primitive types), they represent the usual arithmetic comparisons.
- For Date, Time, and TimeStamp primitive type operands, < is interpreted as "earlier than", while > is interpreted as "later than". For example, date1 < date2 is interpreted as "date1 is earlier than date2".
- For TimeStampInterval primitive type operands, they represent the usual comparisons of the size of the time intervals.
- For Point primitive type operands, the y values are compared first, followed by the x values. If the y values differ, this determines the result of the comparison. If the y values are equal, the x values are compared, to determine the result.

This mode of comparison yields a left-to-right, top-to-bottom ordering of points, consistent with the usual semantics of screen display locations.

Comparing Numerical Expressions of Different Types

Comparisons can be made between numeric expressions that have a different type, as in the following example.

```jade
vars
    int : Integer;
    dec : Decimal[4,2];
begin
    int := 7;
    dec := 5.2;
    if int > dec then
        write "integer is bigger";
    endif;
end;
```

Before the comparison is made, the expression with the lower numeric type is converted to the type of the other expression.
In the following list of the order of numeric types, the **Byte** primitive type has the lowest order and **Decimal** the highest.

1. **Byte**
2. **Integer**
3. **Integer64**
4. **Real**
5. **Decimal**

In the previous example, the int operand is converted to **Decimal** (the same type as the dec operand) before the comparison is made.

**String Concatenation Operator**

The string concatenation operator is the ampersand symbol (**&**). Use the ampersand (**&**) operator to concatenate strings or binaries; for example:

```jade
greeting(name: String): String;
begin
    return "Hello " & name & ". How are you?";
end;
```

Using this example, execute the following code.

```jade
write greeting("Ricky Alfonso");
```

The execution of this code displays the following in the Jade Interpreter Output Viewer window.

```
Hello Ricky Alfonso. How are you?
```
Tip  A concatenating operation involving strings, binary values, and UTF8 strings is optimized in JADE when the values being concatenated are local variables, constants, or literals. In this case, the JADE interpreter determines the size of the resulting string, allocates a single block in the string pool for the result, and constructs the concatenated value directly into this block.

If the values being concatenated are properties or the results of method calls, the operation is more complex and multiple blocks are allocated from the string pool. As allocating blocks is a relatively expensive operation, avoiding this significantly improves the elapsed time of a concatenation operation. The following example shows non-recommended and recommended coding techniques.

```java
vars
    prod : Product;
    xml : String;
    code : String;
    desc : String;
begin
    // Relatively inefficient
    foreach prod in products do
        xml := xml & "<product><code>" & prod.code & "</code><desc>" & prod.desc & "</desc></product>";
    endforeach;
    // Using local variables is much more efficient
    foreach prod in products do
        code := prod.code;
        desc := prod.desc;
        xml := xml & "<product><code>" & code & "</code><desc>" & desc & "</desc></product>";
    endforeach;
end;
```

### Substring Operators

Other operators are the [:] or [] substring operators, which operate on strings and binaries to extract or assign substrings or characters.

JADE string handling notation is listed in the following table. (The yield of the string operator notation, using a value of "Hello world" for string1, is given in the comment after the examples.)

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[n]</td>
<td>String index, returning the single character at the position specified by the n value in the string; for example:</td>
</tr>
<tr>
<td></td>
<td>char := string1[11]; // char = &quot;d&quot;</td>
</tr>
<tr>
<td>[n:length]</td>
<td>Returns the substring from the position specified by the n value for the length specified by the length value in the string; for example:</td>
</tr>
<tr>
<td></td>
<td>string2 := string1[5:4]; // string2 = &quot;o wo&quot;</td>
</tr>
<tr>
<td>[n:end]</td>
<td>Returns the substring from the position specified by the n value to the end of the string; for example:</td>
</tr>
<tr>
<td></td>
<td>string2 := string1[5:end] // string2 = &quot;o world&quot;</td>
</tr>
</tbody>
</table>

**Note** The substring start and length values must be of type `Integer`. The other numeric types, such as `Byte` and `Integer64`, are not allowed in this context.
For details about the use of subscript operators, see "Using Subscripts in Dictionaries and Arrays", in the following subsection.

**Using Subscripts in Dictionaries and Arrays**

The bracket ([ ]) subscripting operators enable you to assign values to and retrieve values from an array or a dictionary.

The code fragments in the following examples show the syntax of bracket subscripting operators for arrays and dictionaries.

```plaintext
stringArray[5] := "five";
str := stringArray[5]; // str = "five"
productDict[prodName] := prod;
prod := productDict[prodName]; // obtain product at key prodName
customerDict["Sid Who", "12 Any Avenue", date1] := cust;
cust := customerDict["Sid Who", "12 Any Avenue", date1];
objArray[row - 1] := obj;
name[int : 1] := ' ';
str := "Hello$$$$World";
write str; // outputs Hello$$$$World
str[6:3] := "***";
write str; // outputs Hello***World
```

**Note** The index value for an array must be of type Integer. The other numeric types, such as Byte and Integer64, are not allowed in this context.

**Operator Precedence**

A JADE expression is evaluated according to the notion of operator precedence, as with normal mathematical expressions. Each of the operators is assigned precedence.

In the following table, operators are listed in order of decreasing precedence (from closest to farthest binding).

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>. (dot notation for properties, instance methods, and casts), @ (at notation for type methods)</td>
</tr>
<tr>
<td>8</td>
<td>^ (exponentiation, or power of)</td>
</tr>
<tr>
<td>7</td>
<td>+ - (unary)</td>
</tr>
<tr>
<td>6</td>
<td>* / mod div</td>
</tr>
<tr>
<td>5</td>
<td>+ - (binary)</td>
</tr>
<tr>
<td>4</td>
<td>= &lt;&gt; &gt;= &lt;= &lt; &gt;</td>
</tr>
<tr>
<td>3</td>
<td>not</td>
</tr>
<tr>
<td>2</td>
<td>and</td>
</tr>
<tr>
<td>1</td>
<td>or</td>
</tr>
</tbody>
</table>
Operators at the same level associate from left to right, except for the level-4 relational operators, which are not associative.

**Type Casts**

JADE provides two distinct forms of type cast operations, each using the dot operator (.) notation. These forms are:

- Primitive type conversions (for details, see "Converting Primitive Types", later in this section)
- Type guards or guarded expressions (for details, see "Type Guard Expression", later in this section)

The language-intrinsic type conversions are deliberately locale-independent so that they produce the same result on any node. This enables you to write code in a locale-neutral manner, to ensure it produces the same results regardless of the locale. For details about:

- Control-specific methods relating to primitive type conversion, see methods of the TextBox and JadeEditMask control classes.
- Locale-dependent methods of primitive types (for example, the Date::format method), see the appropriate primitive type in Chapter 1 of the JADE Encyclopaedia of Primitive Types.
- Creating user-defined formats, see the LocaleFormat class and its subclasses.

**Converting Primitive Types**

You can convert any variable of a primitive type to another primitive type, by using the following syntax.

```
expression.primitive-type
```

The following example shows the conversion of a Date primitive type to a String primitive type.

```
writeDate(date: Date);
begin
  write "The date is " & date.String;
end;
```

**Notes** As the changing of collection membership of primitive types is not supported by the reorganization process, a warning message is displayed when a collection membership of primitive types is changed from one primitive type to another and the class is already marked for reorganization or the change will cause the class to be marked for reorganization.

Converting any primitive value to a Binary primitive type obtains the internal representation of that value. Conversion of this intermediate Binary value to any type other than the original type may produce a different value from a direct conversion. For this reason, converting a primitive value from one type to another using an intermediate Binary value is not recommended. See also "Binary Primitive Type Conversions".

The following example shows an Integer primitive type being converted to a Real primitive type through an intermediate Binary value.

```
vars
  i : Integer;
begin
  i := 10;
  write i.Real;    //outputs 10
  write i.Real;    //outputs 4.94065645841247e-323  Not recommended
end;
```
The following table lists the effect of converting to the **Byte** and **Character** primitive types. Conversions from the **MemoryAddress**, **Point**, **Date**, **Time**, **TimeStamp**, **TimeStampInterval**, and **TimeStampOffset** types are not permitted.

<table>
<thead>
<tr>
<th>From</th>
<th>To Byte</th>
<th>To Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>The first byte is copied.</td>
<td>In a Unicode system, the first two bytes of the <strong>Binary</strong> are taken as a <strong>Character</strong> (null-padded, if necessary). In an ANSI system, the first byte of the <strong>Binary</strong> is taken as a <strong>Character</strong>.</td>
</tr>
<tr>
<td>Boolean</td>
<td>If true, the character is converted to 1; false is converted to zero (0).</td>
<td>If true, the character with ANSI or Unicode value equal to 1; if false, the <strong>Character</strong> with ANSI or Unicode value equal to zero (0).</td>
</tr>
<tr>
<td>Byte</td>
<td>No change.</td>
<td>Character with ANSI or Unicode value equal to the value of the <strong>Byte</strong>.</td>
</tr>
<tr>
<td>Character</td>
<td>Numeric value in the range zero (0) through 255. If the value of the <strong>Character</strong> in a Unicode system is outside this range, an exception is raised.</td>
<td>No change.</td>
</tr>
<tr>
<td>Decimal</td>
<td>Numeric value, truncated if necessary.</td>
<td>Character with ANSI or Unicode value equal to the integer part of the number.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> If the resulting <strong>Byte</strong> value is outside the range zero (0) through 255.9999, an exception is raised.</td>
<td><strong>Note</strong> If the resulting <strong>Character</strong> value is outside the range -255 through 255 ANSI, an exception is raised. For Unicode, the range that can be converted is zero (0) through 65535.9999, with no exception being raised.</td>
</tr>
<tr>
<td>Integer and Integer64</td>
<td>Numeric value (an exception is raised if the value exceeds the maximum size of the target).</td>
<td>Character with ANSI or Unicode value equal to the number.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> If the resulting <strong>Byte</strong> value is outside the range zero (0) through 255, an exception is raised.</td>
<td><strong>Note</strong> If the resulting <strong>Character</strong> value is outside the range zero (0) through 255 ANSI or the range zero (0) through 65535 Unicode, an exception is raised.</td>
</tr>
<tr>
<td>Real</td>
<td>Numeric value, truncated if necessary.</td>
<td>Character with ANSI or Unicode value equal to the integer part of the number.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> If the resulting <strong>Byte</strong> value is outside the range zero (0) through 255, an exception is raised.</td>
<td><strong>Note</strong> If the resulting <strong>Character</strong> value is outside the range zero (0) through 255 ANSI or the range zero (0) through 65535 Unicode, an exception is raised.</td>
</tr>
</tbody>
</table>
From String and StringUtf8 To Byte Numeric value represented by the string of digits. Only US-ASCII characters in the range zero (0) through 9 are recognized.

**Note** The string can contain an optional string of tab and space characters, an optional sign, or a string of numeric values. The first unrecognized character ends the conversion. If the string cannot be converted to a numeric value, JADE returns zero (0), as shown in the following examples.

```
"abc".Byte = 0
"12abc".Byte = 12
```

Overflow exceptions occur if the result of the conversion is too large. Use the `isByte` method to check the contents of a string.

The following table lists the effect of converting to the Boolean primitive type. Conversion from the `MemoryAddress` and `Point` types is not permitted.

<table>
<thead>
<tr>
<th>From</th>
<th>To Boolean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>If all bits in the first byte are zero (0), then <strong>false</strong>; else <strong>true</strong></td>
</tr>
<tr>
<td>Boolean</td>
<td>No change</td>
</tr>
<tr>
<td>Byte</td>
<td>If all bits are zero (0), then <strong>false</strong>; else <strong>true</strong></td>
</tr>
<tr>
<td>Character</td>
<td>If null, then <strong>false</strong>; else <strong>true</strong></td>
</tr>
<tr>
<td>Date</td>
<td>If null, then <strong>false</strong>; else <strong>true</strong></td>
</tr>
<tr>
<td>Decimal</td>
<td>If zero (0), then <strong>false</strong>; else <strong>true</strong></td>
</tr>
<tr>
<td>Integer and Integer64</td>
<td>If zero (0), then <strong>false</strong>; else <strong>true</strong></td>
</tr>
<tr>
<td>Real</td>
<td>If zero (0), then <strong>false</strong>; else <strong>true</strong></td>
</tr>
<tr>
<td>String and StringUtf8</td>
<td>If &quot;true&quot;, then <strong>true</strong>; else <strong>false</strong></td>
</tr>
<tr>
<td>Time</td>
<td>If null, then <strong>false</strong>; else <strong>true</strong></td>
</tr>
<tr>
<td>TimeStamp</td>
<td>If null, then <strong>false</strong>; else <strong>true</strong></td>
</tr>
<tr>
<td>TimeStampInterval</td>
<td>If null, then <strong>false</strong>; else <strong>true</strong></td>
</tr>
<tr>
<td>TimeStampOffset</td>
<td>If null, then <strong>false</strong>; else <strong>true</strong></td>
</tr>
</tbody>
</table>
The following table lists the effect of converting to the `String` and `StringUtf8` primitive types.

<table>
<thead>
<tr>
<th>From</th>
<th>To String and StringUtf8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>A direct binary copy is made, which when converted to a <code>StringUtf8</code>, may result in an illegal byte sequence.</td>
</tr>
<tr>
<td>Boolean</td>
<td><code>true</code> is converted to &quot;true&quot;; <code>false</code> is converted to &quot;false&quot;.</td>
</tr>
<tr>
<td>Byte</td>
<td>Binary value of the <code>Byte</code> in a decimal format.</td>
</tr>
<tr>
<td>Character</td>
<td>One-character string.</td>
</tr>
<tr>
<td>Date</td>
<td>Date in the format <code>dd MMMM yyyy</code>. If the receiver has the invalid value, the string is &quot;<em>invalid</em>&quot;. If the receiver is zero (0), an empty string is returned. The resulting string is always limited to ASCII digits and characters, it is never locale-sensitive, and English month names are used.</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal number formatted to the required number of decimal places.</td>
</tr>
<tr>
<td>Integer and Integer64</td>
<td>Integer value.</td>
</tr>
<tr>
<td>MemoryAddress</td>
<td>Memory address in a hexadecimal number format.</td>
</tr>
<tr>
<td>Point</td>
<td>Cartesian coordinates in the format <code>x,y</code>.</td>
</tr>
<tr>
<td>Real</td>
<td>Number displayed in expanded form.</td>
</tr>
<tr>
<td>String and StringUtf8</td>
<td>Typecasting assumes that the string to be converted contains ASCII characters only. Use the <code>asString</code> and <code>asStringUtf8</code> methods for conversions based on a locale in an ANSI system.</td>
</tr>
<tr>
<td>Time</td>
<td>Time in the format <code>hh:mm:ss</code>.</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>Date and time in the format <code>dd MMMM yyyy, hh:mm:ss</code>.</td>
</tr>
<tr>
<td>TimeStampInterval</td>
<td>Timestamp interval in the format <code>d:hh:mm:ss.fff</code>.</td>
</tr>
<tr>
<td>TimeStampOffset</td>
<td>Timestamp offset in the format <code>d:hh:mm:ss.fff +/-hhmm</code>.</td>
</tr>
</tbody>
</table>

The following table lists the effect of converting to the `Integer` and `Integer64` primitive types. Conversions from the `MemoryAddress`, `Point`, `TimeStamp`, `TimeSpanInterval`, and `TimeStampOffset` primitive types are not permitted.

<table>
<thead>
<tr>
<th>From</th>
<th>To Integer and Integer64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>First 4 or 8 bytes are copied as the <code>Integer</code> or <code>Integer64</code> result, respectively. If the <code>Binary</code> value has fewer than 4 bytes or 8 bytes, the converted result is padded with null bytes.</td>
</tr>
<tr>
<td>Boolean</td>
<td><code>true</code> is converted to 1; <code>false</code> is converted to zero (0).</td>
</tr>
<tr>
<td>Byte</td>
<td>Numeric value in the range zero (0) through 255.</td>
</tr>
<tr>
<td>Character</td>
<td>Numeric value in the range zero (0) through 255 for an ANSI system, or zero (0) through 65,535 for a Unicode system.</td>
</tr>
<tr>
<td>Date</td>
<td>Julian day number for <code>Integer</code> and <code>Integer64</code>.</td>
</tr>
<tr>
<td>Decimal</td>
<td>Numeric value, truncated if necessary.</td>
</tr>
<tr>
<td>Integer and Integer64</td>
<td>Numeric value (an exception occurs if the value exceeds the maximum size of the target type).</td>
</tr>
</tbody>
</table>
From | To Integer and Integer64
--- | ---
Real | Numeric value (an exception occurs if the value exceeds the maximum size of the target type).
String and StringUtf8 | Numeric value represented by the string of digits. Only US-ASCII characters in the range zero (0) through 9 are recognized.

**Note** The string can contain an optional string of tab and space characters, an optional sign, or a string of numeric values. The first unrecognized character ends the conversion. If the string cannot be converted to a numeric value, JADE returns zero (0), as shown in the following examples.

```
"abc".Integer = 0
"12abc".Integer = 12
```

Overflow exceptions occur if the result of the conversion is too large. Use the isInteger method or the isInteger64 method to check the contents of a string.

Time | Milliseconds since midnight for Integer and Integer64.

The following table lists the effect of converting to the Decimal and Real primitive types. Conversions from MemoryAddress, Point, TimeStamp, TimeStampInterval, and TimeStampOffset types are not permitted.

<table>
<thead>
<tr>
<th>From</th>
<th>To Decimal and Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>Sufficient bytes are copied to the Decimal or Real result (padded with null bytes, if necessary). The result of the conversion may not be meaningful.</td>
</tr>
<tr>
<td>Boolean</td>
<td>true = 1, false = 0.</td>
</tr>
<tr>
<td>Byte</td>
<td>Numeric value in the range zero (0) through 255.</td>
</tr>
<tr>
<td>Character</td>
<td>Numeric value in the range zero (0) through 255 for an ANSI system, or zero (0) through 65,535 for a Unicode system.</td>
</tr>
<tr>
<td>Date</td>
<td>Julian day number.</td>
</tr>
<tr>
<td>Decimal</td>
<td>Numeric value.</td>
</tr>
<tr>
<td>Integer and Integer64</td>
<td>Numeric value.</td>
</tr>
<tr>
<td>Real</td>
<td>Numeric value.</td>
</tr>
<tr>
<td>String and StringUtf8</td>
<td>Numeric value. Only US-ASCII characters in the range 0 through 9 are recognized.</td>
</tr>
</tbody>
</table>

**Note** The string can contain an optional string of tab and space characters, an optional sign, or a string of numeric values. The first unrecognized character ends the conversion. If the string cannot be converted to a numeric value, JADE returns zero (0), as shown in the following examples.

```
"abc".Real = 0
"1.2abc".Decimal = 1.2
```

If you are uncertain of the contents of a string, call the isDecimal or the isReal method.

Time | Milliseconds since midnight.
The following table lists the effect of converting to the `Date` and `Time` primitive types. Conversions from `Boolean`, `Byte`, `Character`, `MemoryAddress`, `Point`, `TimeStampInterval`, and `TimeStampOffset` types are not permitted.

<table>
<thead>
<tr>
<th>From</th>
<th>To Date</th>
<th>To Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>First 4 bytes are copy as the <code>Date</code> value. If the <code>Binary</code> value has fewer than 4 bytes, null bytes are used for padding. The result of conversion may not be meaningful.</td>
<td>First 4 bytes are copy as the <code>Time</code> value. If the <code>Binary</code> value has fewer than 4 bytes, null bytes are used for padding. The result of conversion may not be meaningful.</td>
</tr>
<tr>
<td>Date</td>
<td>No change.</td>
<td>Not permitted.</td>
</tr>
<tr>
<td>Decimal</td>
<td>Julian day number is converted to <code>Date</code>.</td>
<td>Milliseconds since midnight is converted to <code>Time</code>.</td>
</tr>
<tr>
<td>Integer and Integer64</td>
<td>Julian day number is converted to <code>Date</code>.</td>
<td>Milliseconds since midnight is converted to <code>Time</code>.</td>
</tr>
<tr>
<td>Real</td>
<td>Julian day number is converted to <code>Date</code>.</td>
<td>Milliseconds since midnight is converted to <code>Time</code>.</td>
</tr>
<tr>
<td>String and StringUtf8</td>
<td>Valid date string is converted to <code>Date</code>.</td>
<td>Valid time string <code>hh:mm:ss.fff</code> is converted to <code>Time</code>. (For more details, see the note following this table.)</td>
</tr>
<tr>
<td>Time</td>
<td>Not permitted.</td>
<td>No change.</td>
</tr>
<tr>
<td>TimeStamp</td>
<td><code>Date</code> part copied, <code>Time</code> part discarded.</td>
<td><code>Time</code> part copied, <code>Date</code> part discarded.</td>
</tr>
</tbody>
</table>

When casting a string to a date, the following rules apply.

- The `String` must contain the date elements in one of the following orders.
  - `<day><month><year>`
  - `<month><day><year>`
  - `<year><month><day>`

- The delimiter between date elements is a sequence of one or more non-alphanumeric characters.

- The day and year elements must be numeric. If the month element is not numeric, it must be a short (three-character) or long English month name (case-insensitive).

- If the year element is one or two digits, it is assumed to be from the current century.

- If the first element is numeric, it is assumed to be a year if it has:
  - Four or more digits
  - Two or more digits and the first separator begins with a colon (·)
The following table lists the effect of converting to the `TimeStamp` and `TimeStampInterval` primitive types. Conversions from the `Boolean`, `Byte`, `Character`, `Decimal`, `Integer`, `Integer64`, `MemoryAddress`, `Point`, `Real`, and `TimeStampOffset` types are not permitted.

<table>
<thead>
<tr>
<th>From</th>
<th>ToTimeStamp</th>
<th>ToTimeStampInterval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>First 8 bytes are copied as the <code>TimeStamp</code> value. If the <code>Binary</code> value has fewer than 8 bytes, null bytes are used for padding. Result of conversion may not be meaningful.</td>
<td>First 8 bytes are copied as the <code>TimeStampInterval</code> value. If the <code>Binary</code> value has fewer than 8 bytes, null bytes are used for padding. Result of conversion may not be meaningful.</td>
</tr>
<tr>
<td>Date</td>
<td><code>Date</code> copied, <code>Time</code> null.</td>
<td>Not permitted.</td>
</tr>
<tr>
<td>String and StringUtf8</td>
<td><code>Date</code> and <code>Time</code> string converted to <code>TimeStamp</code>.</td>
<td>In the format <code>ddd:hh:mm:ss.fff</code>.</td>
</tr>
<tr>
<td>Time</td>
<td><code>Date</code> null, <code>Time</code> copied.</td>
<td>Not permitted.</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>No change.</td>
<td>Not permitted.</td>
</tr>
<tr>
<td>TimeStampInterval</td>
<td>Not permitted.</td>
<td>No change.</td>
</tr>
</tbody>
</table>

**Binary Primitive Type Conversions**

The result of a conversion of any type to a `Binary` primitive type is a binary representation of the source item.

A conversion from a `Binary` primitive type to another type takes the first \( n \) bytes of the `Binary` item, where the \( n \) value is the size of the target type. If the `Binary` item is shorter than the target item, the target item is padded with zeros.

**Note** Converting `MemoryAddress` and `Point` primitive types to primitive types other than the `Binary` type are not permitted.
Type Guard Expression

The syntax of the type guard expression is:

\[
\text{object-reference-expression.class-name}
\]

When a type guard is applied to an object reference, the reference is treated as a reference to an instance of the class specified by the \text{class-name} value. If the expression is not of class \text{class-name}, a runtime exception is raised. For more details, see "Using Type Guards", in the following subsection.

Using Type Guards

A type guard can be required in two cases, as follows.

- When assigning to an object reference where the declared class of the expression on the right of the assignment is a superclass of the declared class on the left-hand side; for example:

```plaintext
countAllClasses(): Integer updating;
 constants
    Max_Classes = 16000;
 vars
    objArray : ObjectArray;
    cls : Class;
    obj : Object;
    total : Integer;
 begin
    create objArray transient;
    Class.allInstances(objArray, Max_Classes, true);
    foreach obj in objArray do
        cls := obj.Class; // Object is a superclass of Class
        if not cls.isSystemObject and cls.superschemaType = null then
            total := total + 1;
        endif;
    endforeach;
    return total;
 end;
```

- When accessing a method or a property of an object where the declared class of the reference to the object is a superclass of the class in which the feature was defined; for example:

```plaintext
foreach form in formDictionary do
    if form.isKindOf(SpecialForm) then
        form.SpecialForm.cutEnabled := true;
    endif;
endforeach;
```

In the above example, a reference is obtained to a form from a generic form dictionary, where the membership type is \text{Form}. To access the \text{cutEnabled} property that is defined in \text{SpecialForm} (a subclass of \text{Form}), a type guard is inserted that asserts that \text{form} is indeed a \text{SpecialForm}.

\textbf{Note}  The use of the \text{isKindOf} method guarantees that the \text{form} reference in the guarded expression is of type \text{SpecialForm}, so a runtime exception will not be raised.
Assignments

This section provides the syntax, description, and an example of JADE assignments, as well as:

- Assignment Compatibility
- Issues and Side Effects
- Object References versus Primitive Values

Syntax

```
left-hand-side := expression;
```

Description

The colon and equals symbols (:=) used in conjunction are known as the assignment operator. The assignment operator is usually read as becomes.

The types of left-hand-side and expression values in the assignment operator must be compatible. The left-hand-side can be one of the following.

- A property expression
- A local variable
- A formal parameter

When an assignment instruction is executed, the expression on the right of the assignment operator is evaluated and assigned to the entity on the left of the assignment operator. If the left-hand-side is a compound expression (for example, `customer.name`), this is evaluated before the assignment.

Example

The following example shows the use of assignments.

```
vars
    anInt : Integer;
    aBool : Boolean;
    aStr : String;
    aReal : Real;
begin
    anInt := 2;
    aBool := true;
    aStr := "a string";
    aReal := 2.0;
end;
```

Assignment Compatibility

Assignment compatibility requires that the type of the expression on the left-hand side of the assignment is assignment-compatible with the expression on the right-hand side.

A value `t2` of type `T2` is assignment-compatible with a value `t1` of type `T1` (that is, `t1 := t2` is allowed) if any of the following is true.

- `T1` is an interface and `T2` is a class that implements the interface
- `T1` and `T2` are identical types
- `T2` is a subclass of `T1`
Chapter 1  JADE Language Reference

- \( T_1 \) is a \texttt{String} primitive type and \( T_2 \) is a \texttt{Character} primitive type
- \( T_1 \) is a \texttt{Real} primitive type and \( T_2 \) is a \texttt{Byte}, \texttt{Integer}, \texttt{Integer64}, or \texttt{Decimal} primitive type
- \( T_1 \) is a \texttt{Decimal} primitive type and \( T_2 \) is a \texttt{Byte}, \texttt{Integer}, \texttt{Integer64}, or \texttt{Real} primitive type
- \( T_1 \) is a \texttt{Integer} primitive type and \( T_2 \) is a \texttt{Byte} primitive type
- \( T_1 \) is an \texttt{Integer64} primitive type and \( T_2 \) is an \texttt{Integer} or \texttt{Byte} primitive type

JADE provides a type-safe mechanism to assign a superclass reference to a declared subclass reference. For details, see "Type Guard Expression", earlier in this chapter.

Issues and Side Effects

The issues and side effects relating to assignments are as follows.

- If an assignment is made to an entity that represents a property of a persistent object, the application must be in transaction state. If the application is not in transaction state, an exception is raised.
- If an assignment is made to a property that is the manual reference of an automatic relationship, the inverse property is updated automatically.
- You cannot assign to the automatic reference of an automatic relationship.
- If an assignment is made to a property that is the key of a collection, the object is automatically removed from the collection at the old key value and added again at the new key value.

\textbf{Note}  JADE aborts the current transaction if it detects an error during an automatic update (for example, updating an inverse reference or a key).

Object References versus Primitive Values

JADE distinguishes between \textit{object-references} (references to objects that are instances of a class or interface) and \textit{primitive-values} (instances of primitive types such as \texttt{Integer} or \texttt{String}). There is a subtle distinction in the way that an assignment behaves for these two types of reference, as follows.

- The assignment of object references uses reference semantics. When one object reference is assigned to another, the object itself is not copied but instead both variables now refer to the same object, as shown in the following example.

```plaintext
vars
    fred, joe : Customer;
begin
    create fred transient;
    fred.name := "Fred";
    joe := fred;  // joe shares a reference to fred
    joe.name := "Joe";
end;
```

In the previous example, as the variables \texttt{joe} and \texttt{fred} both refer to the same object, the statement \texttt{joe.name := "Joe";} also changes the name of \texttt{fred}.

- The assignment of primitive values uses value semantics. The actual value is copied to the target variable and is therefore completely independent of the original value, as shown in the following example.

```plaintext
vars
    int1, int2 : Integer;
```
Epilogs

The epilog, an optional unit of code, is always executed before the method is exited, regardless of whether the method terminates normally or is terminated abnormally as a result of an exception (excluding fatal system terminations, power failures, and so on, that leave the JADE runtime environment inoperable).

Use a method epilog in situations where resources must be relinquished or the state of various objects must be restored to some stable state before the method is exited; for example, when closing files, deleting transient objects, or restoring the cursor from an hourglass to a normal pointer.

You can also use epilogs for the normal processing of a method. For example, use an epilog for cleanup logic so that a method does not contain a number of return instructions with the same cleanup logic repeated before each return instruction; that is, your cleanup logic would appear in only the epilog.

**Tip** When using an epilog to delete an objects, it is not necessary to test for non-null object references before calling the delete instruction.

Define an optional method epilog after the normal method body. As the epilog block has the same scope as the method to which it is attached, it has access to the parameters, constants, and local variables of the method, and to properties and methods of the receiver (self).

There are no restrictions on the kinds of instructions or expressions that may be used in the epilog.

The following example shows the use of the epilog in a print method.

```plaintext
begin
    [method body]
    epilog
    delete tempArray;
    app.printer.close;
    form.unloadForm;
end;
```

If both the method body and the epilog contain return instructions that each return a different value, the value returned by the epilog takes precedence, as it is the last value returned.

If an exception occurs while an epilog is being executed, the execution of the entire epilog cannot be guaranteed; the remainder of the epilog may be aborted or resumed, depending on how the exception is handled. Because of this, you should restrict your epilog logic to actions that are unlikely to result in an exception.
Iterative Instructions

Iterative instructions, or statements, are used to construct a loop that enables the repetition of a sequence of instructions, subject to some terminating condition.

JADE provides two repetition constructs that reflect the needs of loop construction and iterative access to collections. These repetition constructs are listed in the following table.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>while</td>
<td>Constructs a loop subject to a terminating condition</td>
</tr>
<tr>
<td>foreach</td>
<td>Accesses entries of a collection sequentially until a terminating condition is encountered</td>
</tr>
</tbody>
</table>

**while Instruction**

The **while** instruction constructs a loop subject to a terminating condition.

**Syntax**

The syntax of the **while** instruction is:

```
while condition do [:label]
  [while-instructions]
endwhile [label];
```

**Description**

The **condition** is an expression that must produce a value of type **Boolean**. The condition is evaluated repeatedly.

While the condition remains **true**, the optional **while-instructions** are executed following the evaluation of the condition. The loop terminates when the value of the condition is evaluated as **false**.

If the condition is initially **false**, the optional **while-instructions** are not executed. (For details about evaluating a condition once at the start of a loop, see "Iterating over a Range of Numeric Values", under "foreach Instruction", later in this chapter.)

You can nest **while** instructions within each other to any level. The **break** instruction can be used to terminate a **while** loop prematurely or the **continue** instruction can be used to cause the next iteration of a **while** loop to begin.

The optional **:label** identifier is used to nominate the specific loop (within a group of nested loops) that may be terminated or continued by a **break** or **continue** instruction, respectively. When the **break** or **continue** instruction is used without a label, only the innermost loop is terminated or continued. For more details, see "break Instruction" or "continue Instruction", later in this chapter.

**Example**

The following example shows the use of the **while** instruction.

```
writeCustomers(customerArray: CustomerArray);
vars
  count : Integer;
  cust : Customer;
begin
  count := 1;
  while count <= customerArray.size do
    cust := customerArray[count];
    write "Customer " & count.String & ":" & cust.name;
    count := count + 1;
```
Terminating a while Iteration

A while loop can be terminated in the following ways.

- The value of the condition becomes false
- A break or return instruction is executed

foreach Instruction

The foreach instruction accesses entries of a collection sequentially, starting with the first entry and continuing with successive entries until the iteration is terminated.

This section provides the syntax, description, and an example of the foreach instruction, as well as:

- Iterating over a Range of Numeric Values
- Terminating a foreach Iteration
- where Clause Optimization
- Using Iterators in Collections

Syntax

The syntax of the foreach instruction is:

```
foreach variable in collection-expression [options]
   [where condition] do [:label]
   [foreach-instructions]
endforeach [:label];
```

You can specify the reversed or discreteLock option, or both reversed and discreteLock. If you specify both options, you must separate the values with a comma. See also "Iterating over a Range of Numeric Values", later in this section, for details about the collection-expression syntax when executing a sequence of statements a number of times.

Description

You can nest foreach instructions within each other to any level.

The reversed option enables elements of a collection to be accessed in a reverse direction, starting with the last entry and then successive prior entries until the first element is reached. (For details of iterating backwards through collections, see "Using Iterators in Collections", later in this chapter.)

The discreteLock option specifies that a shared lock is not retained for the duration of the foreach instruction. The collection is share locked only for the period during which the foreach instruction is retrieving a snapshot of entries from the associated collection.

The optional where clause, if present, contains a Boolean expression that must evaluate to true to allow the optional foreach-instructions to be executed.
Chapter 1  JADE Language Reference

**Note** If the `discreteLock` option is not specified, execution of a `foreach` iteration attempts to acquire an implicit shared lock on the collection being iterated, so the `foreach` may wait for concurrent users with exclusive locks on the collection. The shared lock ensures that concurrent users cannot add or delete entries from the collection for the duration of the `foreach` loop. The lock is released at the end of the `foreach` loop.

As a `foreach` instruction implicitly locks a collection that is being iterated, if you attempt to add or remove collection members from a `serverExecution` method while in a `foreach` loop on a client method that iterates through the collection, an exception is raised.

The `break` instruction can be used to terminate a `foreach` loop prematurely or the `continue` instruction can be used to cause the next iteration of a `foreach` loop to begin.

The optional `:label` identifier is used by `break` instructions and `continue` instructions to nominate the specific loop that is to be terminated or continued in a group of nested `foreach` loops. For more details, see "break Instruction" or "continue Instruction", later in this chapter.

**Examples**

The following example uses a `foreach` loop to write out the names and ages of all employees in a supplied employee dictionary. In this example, the `calculateAge` is a method on the `Employee` class that returns an integer value of the employee's age.

```jade
writeEmployees(employees: EmployeeDict);
vars
    emp : Employee;
begin
    foreach emp in employees do
        write emp.name & "", " & emp.calculateAge.String & " years old";
    endforeach;
end;
```

The following example uses a `foreach` loop with a `where` clause to write out the names of all employees in a supplied employee dictionary who are older than a specified age.

```jade
writeEmployeesOlderThanAge(employees: EmployeeDict;   
    minimumAge: Integer);   
vars
    emp : Employee;
begin
    foreach emp in employees where emp.calculateAge > minimumAge do
        write emp.name;
    endforeach;
end;
```

**Iterating over a Range of Numeric Values**

For numeric values (that is, `Integer`, `Real`, `Decimal`, `Date`, `Time`, and `TimeStamp` primitive types), you can specify the `collection-expression` of the `foreach` instruction in the following form when iterating over a range of numeric values.

```jade
foreach variable in initial-expression to final-expression
    [step increment-expression] [options]
    [where condition] do [:label]
        [foreach-instructions]
    endforeach [:label];
```
A sequence of instructions is executed once for each value of the loop control variable in a discrete range of numbers.

The variable, increment-expression, initial-expression, and final-expression values must all be numeric.

These parts of the foreach instruction syntax are listed in the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>Contains the value in the range for each iteration of the loop. The variable must be one of the following single lowercase identifiers: the name of a local variable, the name of a parameter in the current method, or an unqualified property name (that is, a property defined in the same class as the method containing the foreach instruction).</td>
</tr>
<tr>
<td>initial-expression</td>
<td>Defines the initial value of the variable.</td>
</tr>
<tr>
<td>final-expression</td>
<td>Defines the final value of the variable.</td>
</tr>
<tr>
<td>increment-expression</td>
<td>Defines the value to be added to variable at the end of each iteration of the loop.</td>
</tr>
</tbody>
</table>

The increment-expression value defaults to 1 but can be specified as a positive or negative numeric expression. The loop continues executing while the value of variable is one of the following.

\[
\begin{align*}
\text{variable} & \leq \text{final-expression (if increment-expression} \geq 0) \\
\text{variable} & \geq \text{final-expression (if increment-expression} < 0)
\end{align*}
\]

The variable can be of type Byte, but if the value of the final-expression plus the value of the increment-expression exceeds 255, an exception is raised, as shown in the following example.

```pascal
vars
  byte: Byte;
begin
  foreach b in 1.Byte to 255.Byte step 1 do
    endforeach;
end;
```

**Note** For performance reasons, the initial, final, and increment expressions are evaluated once only, before entering the loop.

This form of the foreach instruction has the same meaning as the following code sequence, in which incrementExpression >= 0.

```pascal
variable := initialExpression;
finalValue := finalExpression;
incrementValue := incrementExpression;
while variable <= finalValue do
  ...
  // foreach statements
  variable := variable + incrementValue;
endwhile;
```

In this code sequence, finalValue and incrementValue are expressions of the same type as variable. If the value of incrementExpression is less than 0, the while instruction would be as follows.

```pascal
while variable >= finalValue do
```

The following code sequence uses the reversed option.

```pascal
foreach variable in expr1 to expr2 step expr3 reversed do
```
This code sequence using the reversed option is equivalent to the following code sequence.

```java
foreach variable in expr2 to expr1 step -expr3 do

For examples of iterating over a range of numeric values, see "Examples of Numeric Value Iteration", in the following subsection.

Examples of Numeric Value Iteration

The following example of a range of Integer primitive type values displays the integers in the range 1 through 10 in ascending order.

```java
vars
count : Integer;
begin
    foreach count in 1 to 10 do
        write count; // Outputs 1 2 3 4 5 6 7 8 9 10
    end;
end;
```

Use the reversed option to indicate that the range of numbers is to be assigned to the variable in reverse order. The following example displays the integers in the range 1 through 10 in descending order.

```java
vars
count : Integer;
begin
    foreach count in 1 to 10 reversed do
        write count; // Outputs 10 9 8 7 6 5 4 3 2 1
    end;
end;
```

The following example displays only the even integers in the range 1 through 10 in descending order.

```java
vars
count : Integer;
begin
    foreach count in 10 to 1 step -2 do
        write count; // Outputs 10 8 6 4 2
    end;
end;
```

The following example, using the reversed option and the where clause, displays in descending order the integers in the range 1 through 10, except for 5.

```java
vars
count : Integer;
begin
    foreach count in 1 to 10 reversed where count <> 5 do
        write count; // Outputs 10 9 8 7 6 4 3 2 1
    end;
end;
```

Terminating a foreach Iteration

A foreach loop can be terminated in the following ways.

- All collection entries have been accessed
- A break or return instruction is executed
where Clause Optimization

The JADE compiler and interpreter optimize iterations for a dictionary with a single key, provided that the iteration contains a simple relative key expression in the where clause of the foreach instruction.

Note The where clause is not optimized if the dictionary key is sorted in descending order.

The optimization consists of recognizing a relative key expression, to start the iteration at the required relative position in the associated dictionary, as follows.

- In the forward direction, the expressions key > value or key >= value are optimized.
- In the reverse direction, the expressions key < value or key <= value are optimized.

The code fragment in the following example shows the where clause with a relative key expression.

```
foreach customer in company.allCustomers where customer.name > selectedName do
    addToList(customer);
endforeach;
```

In this example, customer.name is the key of the allCustomers dictionary on company.

Using Iterators in Collections

In some cases, the foreach instruction will not be appropriate for your collection handling requirements. For example, if you have a requirement to iterate through two collections simultaneously, the foreach instruction would not allow you to do this. In this case, you would need to use an Iterator object to read through a collection. For more details, see "Iterator Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

The following example uses an iterator to write out the name and ages of all employees in a supplied Employee dictionary.

```
writeEmployees(employees: EmployeeDict);
vars
    emp : Employee;
    iter : Iterator;
begin
    iter := employees.createIterator;
    while iter.next(emp) do
        write emp.name & ", " & emp.calculateAge.String & " years";
    endwhile;
    delete iter;
end;
```

Conditional Instructions

It is often necessary to make the execution of JADE instructions dependent upon a condition or at some point to choose to execute one of a sequence of instructions from several selections.

The conditional instructions provided by JADE are listed in the following table.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>executeWhen</td>
<td>Loads and executes JADE instructions if a Boolean flag value is true</td>
</tr>
<tr>
<td>if</td>
<td>Executes JADE instructions dependent upon a condition</td>
</tr>
</tbody>
</table>
executeWhen Instruction

The executeWhen instruction loads and executes JADE instructions if a Boolean flag value is true.

Syntax

The syntax of the executeWhen instruction is as follows.

```plaintext
executeWhen [not] FlagName;  
optionalStatementList  
endExecuteWhen;
```

The FlagName value is a user-defined Boolean global constant. The value of FlagName can be set for the node in the JADE initialization file or through code at run time.

Description

The executeWhen instruction enables blocks of code in a method to be:

- Not loaded if the value of a boolean flag value is false
- Executed only when the value of a boolean flag value is true

The value of the boolean flag can be:

- Set when the node starts up from a setting in the JADE initialization file
- Changed through code at run time

You could use the executeWhen instruction to bracket debug instructions that are useful during development but not in production, for example. In the production system, the debug instructions are not loaded when the method is loaded prior to execution; that is, only code to be executed is loaded.

Consider loading the following code fragment.

```plaintext
instruction #1;  
executeWhen FlagName;  
instruction #2;  
endExecuteWhen;  
instruction #3;
```

If the effective value of FlagName for the node is true, the instructions loaded are:

```plaintext
instruction #1;  
instruction #2;  
instruction #3;
```

If the effective value of FlagName for the node is false, the instructions loaded are:

```plaintext
instruction #1;  
instruction #3;
```

FlagName is a Boolean global constant that you must have defined

The executeWhen instruction does not examine the true or false value defined when the global constant was created. This value is always ignored by the executeWhen instruction. Instead, the effective value of the global constant is read from the FlagName parameter in the [JadeExecuteFlags] section of the JADE initialization file when the node is initialized (although it can also be changed at run time), as shown in the following example.

```plaintext
[JadeExecuteFlags]  
DebugTestFlag=true
```
If the [JadeExecuteFlags] section does not contain a parameter corresponding to a global constant, the effective value used in an `executeWhen` instruction is `false`.

**Note** The defined value of the global constant is not used and never changes, but its effective value used in an `executeWhen` instruction can change.

The `setExecuteFlagValue` method defined in the `Node` class enables you to change the effective value of the `executeWhen` flag for the current node, thereby overriding the value set in the `FlagName` parameter in the [JadeExecuteFlags] section of the JADE initialization file on that node, as shown in the following code fragment.

```plaintext
node.setExecuteFlagValue("FlagName", false);
node.clearMethodCache;
```

**Note** In this example, the `clearMethodCache` method of the `Node` class is called to discard methods previously loaded into method cache. The `executeWhen` instructions are re-evaluated when methods are reloaded.

The `Node` class also provides the `getExecuteFlagValue` method, which returns the current, effective value of a global constant used in an `executeWhen` condition for the current node.

The `setExecuteFlagValue` method dynamically sets the in-memory value of the `executeWhen` flag for the node on which the method is executed. If the flag needs to be set on another node (for example, the database server), a `serverExecution` method must be called. Note, however, that the method cache must be flushed so that methods are reloaded with the new flag value in effect.

The value of the flag is not re-evaluated (that is, re-read from the initialization file) after the `clearMethodCache` method has been called. The current value is applied to each method as it is reloaded from the database.

The parameter values in the [JadeExecuteFlags] section of the JADE initialization file are loaded into memory the first time:

- An `executeWhen` instruction is encountered while loading a method.
- The `Node::getExecuteFlagValue` method is called.
- The `Node::setExecuteFlagValue` method is called.

The values are not read again until the node is restarted.

**Example**

The following example shows the `executeWhen` instruction used to conditionally load a debug `write` instruction for execution.

```plaintext
executeWhen DebugTestFlag;
   write cust.balance;
endExecuteWhen;
```

**if Instruction**

The `if` instruction executes JADE instructions dependent upon a condition.

**Syntax**

The forms of the JADE conditional instruction are:

- `if boolean-expression then
  [instructions]
endif;`
This executes the instructions if the condition evaluates to **true**.

- \[
\text{if boolean-expression then} \\
\quad [\text{instructions}] \\
\text{else} \\
\quad [\text{instructions}] \\
\text{endif;}
\]

This executes the first sequence of instructions if the condition evaluates to **true** and the second sequence if the condition evaluates to **false**.

- \[
\text{if boolean-expression then} \\
\quad [\text{instructions}] \\
\text{elseif boolean-expression then} \\
\quad [\text{instructions}] \\
\text{elseif boolean-expression then} \\
\quad [\text{instructions}] \\
\text{else} \\
\quad [\text{instructions}] \\
\text{endif;}
\]

The **elseif** clause is equivalent to an **else** clause followed by a further nested **if** instruction. The **else** part remains optional when there is one or more **elseif** clause.

**Description**

Use the **if** instruction to perform conditional execution, based on the evaluation of an expression.

**Examples**

The following example shows a series of nested **if** instructions.

```plaintext
getJobTitle(level: Integer): String;
begin
  if level = 1 then
    return "CEO";
  else
    if level = 2 then
      return "Project Manager";
    else
      if level = 3 then
        return "Analyst";
      else
        if level = 4 then
          return "Programmer";
        else
          return "Invalid level";
        endif;
      endif;
    endif;
  endif;
end;
```

This can be written in a more compact and readable way, by making use of the **elseif** clause, as shown in the following example.

```plaintext
getJobTitle(level: Integer): String;
begin
  if level = 1 then
```
Other Control Structures

JADE provides the control structures listed in the following table.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>break</td>
<td>Breaks out of a loop</td>
</tr>
<tr>
<td>continue</td>
<td>Causes the next iteration of a loop to begin</td>
</tr>
<tr>
<td>return</td>
<td>Terminates execution of a method</td>
</tr>
<tr>
<td>inheritCreate</td>
<td>Invokes the superclass constructor with parameters</td>
</tr>
<tr>
<td>inheritMethod</td>
<td>Invokes the superclass implementation of a method</td>
</tr>
<tr>
<td>terminate</td>
<td>Terminates all active methods</td>
</tr>
<tr>
<td>call</td>
<td>Calls an external function in a library</td>
</tr>
<tr>
<td>importMethod</td>
<td>Calls an imported method</td>
</tr>
</tbody>
</table>

This section also covers method invocation. For details, see "Method Call", later in this chapter.

**break Instruction**

The **break** instruction breaks out of a loop.

**Syntax**

The syntax of the **break** instruction is:

```
  break [label];
```

**Description**

More precisely, a **break** instruction passes control to the first instruction following the innermost matching loop end.

If you specify a label, then control passes to the first instruction following the end matching the label.

**Examples**

The code fragment in the following example shows the use of the **break** instruction to conditionally terminate a **foreach** iteration.

```
foreach count in 1 to self.length do
  if self[count] <> " " then
    break;
```
strOut := self[count : self.length - count + 1];
break;
endif;
endforeach;

The following code fragment is another example of the use of the break instruction to conditionally terminate a foreach iteration.

foreach task in tasklist do
  if task.scheduled then
    break;
  endif;
  task := null;
endforeach;
if task <> null then
  runScheduledTask (task);
endif;

The following code fragment is an example of the use of the break instruction with a label to terminate a while loop from a nested foreach instruction.

found := false;
while count < Max_Tables do : outerloop
  wordtable := tables[count];
  foreach word in wordtable do
    if token = word then
      found := true;
      break outerloop;
    endif;
  endforeach;
  count := count + 1;
endwhile outerloop;

Execution of the break instruction in the above example causes termination of the outer while loop and execution continues with the next instruction following the endwhile.

Note The outerloop label identifier that follows the endwhile is optional but if it is present, the JADE compiler checks that it matches the label following the matching do.

continue Instruction

The continue instruction causes the next iteration of the innermost matching while or foreach loop to begin.

Syntax

The syntax of the continue instruction is:

    continue [label];

Description

In the case of a while loop, the continue instruction passes control to the test of the while condition. In a foreach loop, control passes to the point at which the control variable is assigned its next value.

If you specify a label, control passes to the next iteration of the loop instruction matching the label.

The continue instruction alleviates the need for code within loops becoming too deeply nested. For example, consider a loop that reads some input data, checks that it is valid, and if so, processes it.
If there are a number of validity tests or they are complex, the code may look like that shown in the code fragment in the following example.

```plaintext
while moreData do
    readData;
    if checkFirstDataItem then
        if checkSecondDataItem then
            ...
            if checkLastDataItem then
                processData;
            endif;
            ...
        endif;
    endif;
endwhile;
```

The following example shows the code in the above example rewritten without the deep indentation, by using the `continue` instruction.

```plaintext
while moreData do
    readData;
    if not checkFirstDataItem then
        continue;
    endif;
    if not checkSecondDataItem then
        continue;
    endif;
    ...
    if not checkLastDataItem then
        continue;
    endif;
    processData;
endWhile;
```

**Example**

The following example displays the integers 1 through 10 except for 5 in ascending order, by using the `continue` instruction.

```plaintext
vars
    count : Integer;
begin
    foreach count in 1 to 10 do
        if count = 5 then
            continue;
        endif;
        write count;     // Outputs 1 2 3 4 6 7 8 9 10
    endforeach;
end;
```

**return Instruction**

The `return` instruction terminates execution of the method and optionally return the expression value as the result of the method.
Syntax

The syntax of the `return` instruction is:

```
return [expression];
```

Description

When a `return` instruction includes an expression to be returned, the type of that expression must be compatible with the return type defined in the method signature.

Examples

The following example shows the use of the `return` instruction to return a `StringArray` value.

```java
getAllPositions(): StringArray;
vars
    stringArray : StringArray;
begin
    create stringArray transient;
    stringArray.add ("CEO");
    stringArray.add ("Project Manager");
    stringArray.add ("Analyst");
    stringArray.add ("Analyst Programmer");
    stringArray.add ("Programmer");
    return stringArray;
end;
```

The following example shows the use of the `return` instruction with no returned results.

```java
if customerName = null then
    app.msgBox("A name must be entered", "Error", MsgBox_OK_Only);
    return;
endif;
```

The following example shows the use of the `return` instruction to return a `Boolean` value.

```java
isOpen(): Boolean;
begin
    return closedDate = null;
end;
```

**inheritCreate Instruction**

The `inheritCreate` instruction invokes the superclass constructor with parameters implementation of a method.

Syntax

The syntax of the `inheritCreate` instruction is:

```
inheritCreate (parameters);
```

Description

When a subclass reimplements a method defined in a superclass, you can use the `inheritCreate` instruction from a subclass method to cause execution of the superclass method.

Superclass constructors with parameters are called explicitly from the constructor of any immediate subclasses, using the `inheritCreate` instruction.
It is recommended that `inheritCreate` is the first instruction in the method. However, this is not enforced because it is trivial to write code that executes against a partially constructed object; for example, by using the result of a method call as the value to a parameter to a superclass constructor.

If the superclass constructor has parameters, the compiler will verify that the call to the constructor is coded and that the parameters are correct. If the superclass constructor does not have parameters, an explicit call is not allowed.

In the following hierarchy, when an instance of B is created, A::create is called, then B::create is called with the parameters provided in the create instruction. When an instance of C is created, A::create is called, then C::create. B::create must be called explicitly by C::create.

```
// class hierarchy
A - create() // explicit
 \   B - create(i:Integer) // explicit
    \ C - create() // explicit
A::create();
begin
eend;
B::create(i: Integer);
begin
   // must not call A::create();
eend;
C::create();
begin
   // must call B::create();
   inheritCreate(123);
eend;
```

**Example**

The following method indicates whether the superclass constructor has been called (it should not have been).

```
create() updating;
begin
   createC71 := createC7;
   inheritCreate("football");
eend;
```

The following method is an example of the `inheritCreate` instruction.

```
create() updating;
begin
   inheritCreate("123");
   inheritCreate("456");
eend;
```

See also the `inheritMethod` instruction.

**inheritMethod Instruction**

The `inheritMethod` instruction invokes the superclass implementation of a method.

**Syntax**
The syntax of the `inheritMethod` instruction is:

```
inheritMethod [(parameters)];
```

**Description**

When a subclass reimplements a method defined in a superclass, you can use the `inheritMethod` instruction from a subclass method to cause execution of the superclass method.

If the superclass method takes no parameters, the parentheses are optional.

**Example**

To illustrate the use of the `inheritMethod` instruction, an `openAccount` method in the class `BankAccount` is first written, as follows.

```
openAccount(acName: String;
            acAddr: String;
            acBal: Decimal);
begin
    name := acName;
    address := acAddr;
    balance := acBal;
    status := Account_Status_Open;
end;
```

An `openAccount` method in the class `ChequeAccount` is then written, which first calls the `openAccount` method in the `BankAccount` superclass and then performs specialized processing relevant only to cheque (or check) accounts.

```
openAccount(acName: String;
            acAddr: String;
            acBal: Decimal);
begin
    inheritMethod(acName, acAddr, acBal);
    orderChequeBook;
end;
```

See also the `inheritCreate` instruction.

**Method Call**

The method call (invocation) instruction enables you to send a message to an object.

**Syntax**

The syntax of a method call is:

```
expression.method-name [(actual-parameters)];
```

**Description**

Use a method call to cause the method specified by `method-name` to be invoked for the instance referred to by the expression.

If the method being called takes no parameters, the parentheses on the method call are optional.

**Example**
The following example shows how you can use a method call to invoke the `close` method.

```plaintext
bOK_click(button: Button) updating;
vars
begin
   beginTransaction;
   complaint.close(closeText.text);
   commitTransaction;
   unloadForm;
end;
```

**terminate Instruction**

The terminate instruction terminates all active methods.

**Syntax**

The syntax of the `terminate` instruction is:

```plaintext
terminate;
```

**Description**

The `terminate` instruction cuts back the execution stack and then returns the following result to the controlling application.

```plaintext
APPLICATION_TERMINATED (1209)
```

If the application is a standard JADE application controlled by the `jade.exe` JADE application program, an orderly shutdown of the application is invoked.

**Example**

The following example shows the use of the `terminate` instruction in a method on a form that is used to shut down the application.

```plaintext
shutDownApp();
vars
   mainWindow : Form;
begin
   mainWindow := getMdiFrame;
   if mainWindow <> null then // We have an MDI frame. By unloading it, the app is shut down
      mainWindow.unloadForm;
   else // We don't have an MDI frame, so use terminate to shut down
      terminate;
   endif;
end;
```

**call Expression**

The `call` expression calls an external function in a library by using the Standard calling convention.

**Syntax**

The syntax of the `call` expression is:

```plaintext
call external-function( [actual-parameters] );
```
The external-function identifier can begin with an uppercase or a lowercase character.

**Description**

The parameters of the call expression can be passed by value or by reference. (For details, see "Defining External Functions", in Chapter 8 of the JADE Development Environment User’s Guide. See also "Calling External Functions from JADE Thin Clients", in Chapter 1 of the JADE External Interface Developer’s Reference.)

A call expression can:

- Be used as an instruction in its own right
- Appear as the right-hand side of an assignment
- Be part of a more-complex expression (for example, a call could be used as one of the parameters to a method call)

The JADE external function facility passes parameters using the following conventions.

- Structured parameter (String and Binary) types
  - Always passed by reference; that is, the address of the String or Binary variable is taken and passed
  - When usage is output or io (which signifies the calling function may update the value), length must be specified in the parameter definition
  - A function of String primitive type is assumed to return a zero-terminated string
  - A function of Binary primitive type must have the length of the Binary specified in the return type

- Simple parameter (Integer, Character, Real, Boolean, Time, and Date) types
  - Parameter is passed by value when the usage is defined as constant or input
  - Parameter is passed by reference when the usage is defined as output or io

**Example**

The following example shows the use of the call expression.

```plaintext
getIniParameter(key: String; default: String): String;
vars
    value : String[100];
begin
    call GetProfileString("Widgets", key, default, value, value.maxLength);
    return value;
end;
```

In this example, a fixed-length string is passed to the value parameter. The length of the value parameter string is passed in the length parameter. (This could also have been hard-coded to the length of the value parameter defined in the function signature.)

**importMethod Call**

The importMethod call (invocation) instruction enables you to send a message to an imported method. For details about importing packages, see Chapter 8, "Using Packages".
### Syntax

The syntax of an `importMethod` call is:

\[
\text{expression} := \text{class-instance}.\text{importMethod} \text{ imported-method-name};
\]

\[
\text{expression.type-guard.importMethod} \text{ imported-method-name};
\]

### Description

Use an `importMethod` call to cause the imported method specified by `imported-method-name` to be invoked for the instance referred to by the expression.

If the imported method being called takes no parameters, the parentheses on the imported method call are optional.

In cases where a locally defined method has the same name as an exported method, developers in the importing schema can use the `importMethod` call to specify that it is the imported method that is to be invoked.

When calling an imported method, the following rules apply.

- Given a reference to an instance of an imported class and an imported method on that class has the same name as a locally defined superclass method, you must use the `importMethod` call to invoke the imported method via that reference.

- If `ref` holds a reference to an imported class instance at run time, `ref.m1` always invokes the local `Object::m1` method because locally defined methods are resolved first.

Consider the following conditions.

- A reference, `ref`, of type `Object`
- An imported class, `C`
- A local subschema copy method, `m1`, defined on `Object`
- An imported method, `m1`, defined on `C`

Under these conditions, to invoke the `C::m1` via the `ref` reference, `ref` must be type-guarded to `C` and the `importMethod` call used as follows.

\[
\text{ref.C.importMethod m1;}
\]

The type guard is necessary because the `importMethod` call is permitted only when the type of the receiver, `ref`, is an imported class.

When invoking an `Object` subschema copy method via a reference of type `Object` and that method has the same name as an imported method, binding to the local `Object` method by default is necessary to preserve the behavior of existing code.

### Simple Input and Output Instructions

The JADE input and output instructions are listed in the following table.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>read</strong></td>
<td>Displays the User Input dialog, enabling you to specify the value of the primitive type variable that is required</td>
</tr>
<tr>
<td><strong>write</strong></td>
<td>Outputs the value of an expression to the Jade Interpreter Output Viewer window</td>
</tr>
</tbody>
</table>
**read Instruction**

The read instruction displays the User Input dialog, enabling you to specify the value of the primitive type variable that is required.

**Syntax**

The syntax of the read instruction is:

```plaintext
read primitive-type-variable;
```

The variable can be of any primitive type.

**Description**

The current value of the specified primitive type variable is displayed in the text box, by default. If the variable is of type Any, the input is handled as a String.

The length of data that can be returned when a string is updated by the read instruction is 2048 characters.

You can use the ReadEnabled parameter in the [JadeInterpreter] section of the JADE initialization file to disable the display of the User Input dialog by a read instruction.

To specify a variable value

1. In the text box of the User Input dialog, specify the value for the variable that is being read. For example, if your test method contains the following logic, specify the number that is being tested.

   ```plaintext
   read numberToTest;
   foreach count in 1 to numberToTest do
   ...
   ```

2. Click the OK button.

   Alternatively, click the Cancel button to assign a null value to the variable.

The action that follows the read instruction in your method is then performed.

---

**write Instruction**

The write instruction outputs the value of an expression to the Jade Interpreter Output Viewer window.

**Syntax**

The syntax of the write instruction is:

```plaintext
write expression;
```

The expression value can be of any type.

If you use the write instruction with an operand that is an object reference, the internal object identifier (oid) of that object is displayed.

**Description**

You can use the WriteEnabled parameter in the [JadeInterpreter] section of the JADE initialization file to disable the output of write instructions to the Jade Interpreter Output Viewer.
It is common to build up a formatted string expression to act as the operand of a write instruction. In order to build up a string using expressions of different types, you must use the .String string conversion operator in order to convert non-string operands to strings. For more details, see "Converting Primitive Types", earlier in this chapter.

For details about viewing the output from your executed method, see "Using the Jade Interpreter Output Viewer", in Chapter 1 of the JADE Runtime Application Guide.

Notes Although server methods are prevented from executing code that requires Graphical User Interface (GUI) objects (for example, forms and controls), you can use the Jade Interpreter Output Viewer for write instructions in your JADE applications running on a server. The Interpreter Output Viewer is a separate module that talks directly to the operating system and does not require any GUI objects.

When JADE is running in thin client mode, the write instruction is always executed on the presentation client workstation.

Example

The following example shows the use of the write instruction.

```jade
writeDate(date: Date);
vars
begin
  write "The date today is " & date.format("dd.MM.yyyy");
end;
```

This example, executed on 18 June 2001, outputs the following to the Jade Interpreter Output Viewer.

```
The date today is 18.06.2001
```

Exception Handling Instructions

The JADE exception handling instructions are listed in the following table.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>on exception</code></td>
<td>Arms an exception handler</td>
</tr>
<tr>
<td><code>raise exception</code></td>
<td>Raises an exception</td>
</tr>
</tbody>
</table>

For more details about exception handling, see Chapter 3, "Exceptions".

**on exception** Instruction

The `on exception` instruction arms an exception handler.

**Syntax**

The syntax of the `on exception` instruction is:

```jade
on exception-class do method-call-expression [global];
```

The `exception-class` identifier is the `Exception` class or one of its subclasses. Use the optional `global` system variable to arm a global exception handler. As a global exception handler is not associated with a specific method, it is always armed until it is explicitly disarmed.

A local exception handler is automatically disarmed when the method that armed it returns.
Note Each process may have up to 128 global exception handlers armed at any one time. There is no JADE-imposed restriction on the number of non-global exception handlers that may be armed at any time by any one process.

See also the "Arming Exception Handlers" subsection.

Description

Using the on exception instruction to arm an exception handler causes the specified method to be invoked when an exception of a specified exception class is raised.

Use null in the method-call-expression identifier to disarm an exception handler (using the optional global system variable to disarm a global exception handler).

The following syntax disarms an exception handler.

```
on exception-class do null [global];
```

The signature of the exception handler method must be of the following form.

```
Signature   method-name(e: Exception-class; [other-optional-parameters]) : Integer;
```

In this signature, the e parameter references the exception object created and passed when the exception was raised.

You can optionally specify other parameters when you are creating or arming a method handler to suit your requirements but you must specify the parameter that refers to the exception object as the first parameter. (See also the "Arming Exception Handlers" subsection.) The exception handler would normally use information saved in the Exception class object to decide the action to be taken. Within your method, you can use the isKindOf method to determine the class of the exception object.

An exception handler can also have the following signature.

```
Signature   method-name(exObj: any-exception-class; [other-parameters]) : Integer;
```

In this signature, the exObj parameter indicates that this exception handler handles only exceptions that are instances of the specified exception class or its subclasses. You could specify o and i optional parameters, for example, to indicate the current object and an integer variable, respectively.

Note In these exception handler method signatures, the e or exObj parameter must be the first parameter that you specify in the exception handler method signature.

The contents of parameter expressions passed to the exception handler method are evaluated as follows.

- When the exception is armed for global exception handlers, as shown in the following example.

```
extMethod;
vars
  obj : Object;
  aString : String;
begin
  obj := global;
aString := "very first time";
on Exception do ehWithParameters(exception, aString, obj) global;
aString := "first time";
/* If an exception is caused by the following line of code, the
  aString parameter contains the value "very first time" and the
  obj parameter refers to global */
method2;
```

```
obj := app;
aString := "second time";
/* If an exception is caused by the following line of code, the
aString parameter contains the value "very first time" and the
obj parameter refers to global */
method2;
end;

For local exception handlers, the moment an exception occurs, as shown in the following example.

exampleMethod;
vars
    obj : Object;
    aString : String;
begin
    obj := global;
    aString := "very first time";
    on Exception do ehWithParameters(exception, aString, obj);
    aString := "first time";
    /* If an exception is caused by the following line of code, the
    aString parameter contains the value "first time" and the obj
    parameter refers to global */
    method2;
    obj := app;
    aString := "second time";
    /* If an exception is caused by the following line of code, the
    aString parameter contains the value "second time" and the obj
    parameter refers to app */
    method2;
end;

The return value indicates the action that the system then takes, as shown in the following table.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Constant</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ex_Continue</td>
<td>Continues execution from the next expression after the expression that caused the exception. Use this return mode only in circumstances when you are certain that continuing the code execution will still be correct after ignoring the exception. For lock exceptions, use this return mode only if the lock has been successfully retried. If the lock exception occurred while updating, ensure that the transaction has not been aborted by the exception handler before returning Ex_Continue.</td>
</tr>
<tr>
<td>1</td>
<td>Ex_Abort_Action</td>
<td>Causes the current action to be aborted. The execution stack is stripped back, and in most cases the application reverts to an idle state in which it is waiting for user input or some other Windows event. (For details, see the &quot;Arming Exception Handlers&quot; subsection.)</td>
</tr>
</tbody>
</table>
### Return Value Constants

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Constant</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Ex_Resume_Next</td>
<td>Passes control back to the method that armed the exception handler. Resumes from the next statement after the evaluation of the method call or expression in which the exception occurred. If there were no messages on the execution stack when the handler was armed, the effect of the Ex_Resume_Next call is identical to that of the Ex_Abort_Action call. You cannot resume from global exception handlers. Using this value for a global exception handler is equivalent to returning the Ex_Abort_Action value.</td>
</tr>
<tr>
<td>-1</td>
<td>Ex_Pass_Back</td>
<td>Passes control back to the prior local exception handler for this type of exception or if a local handler is not found, a global exception handler for this type of exception. If neither a local nor a global handler is found, the default exception handler is invoked on a client. On the server, it passes the exception back to the client node. Note that if an exception occurs while executing a server method and no user exception handlers are armed, the default handler is invoked on the server node. This aborts any current transaction on the server and raises a 1242 exception on the client node that invoked the server method execution.</td>
</tr>
</tbody>
</table>

### Examples

The following examples show the use of the `on exception` instruction.

```jade
on SystemException do printException(exception); app.printer.print(folio.detail); foreach portfolio in app.investor.myPortfolios do folio.compName.caption := portfolio.companyName; app.printer.print(folio.compDetail); endforeach;
```

**Note** The `exception` parameter value is a system variable operand that provides a reference to the exception object when an exception is raised. For more information about arming an exception handler, see Chapter 3, "Exceptions".

The following is an example of a method on a `Document` class that writes the document to a file. It arms a local exception handler to catch file exceptions.

```jade
writeToFile(); vars fileSaveDialog : CMDFileSave; file : File; begin on FileException do fileExceptionHandler(exception); create fileSaveDialog transient; if fileSaveDialog.open = 0 then
```
create file transient;
file.mode := File.Mode_Output;
file.fileName := fileSaveDialog.fileName;
saveToFile(file);
file.close;
delete file;
endif;
epilog
delete fileSaveDialog;
end;

Arming Exception Handlers

When you arm an exception handler using the **on exception** instruction, note the following points.

- Any database transaction that is in progress is not aborted.
  You must explicitly code an **abortTransaction** instruction within the exception handler if the database transaction in progress is also to be aborted.

- The **method-call-expression** identifier can be in the form:

  
  object-reference-expression.method-identifier

  In this case, **method-identifier** is invoked for the specified object if an exception occurs.

- The **method-call-expression** identifier can be in the form **method-identifier**, in which case **method-identifier** is invoked for the object upon which the method arming the exception handler is invoked.

- The method used to arm an exception handler can have parameters in addition to the **exception** parameter, which refers to the exception object created and passed when the exception was raised. For example, you could specify an optional parameter that refers to the current object and an optional parameter that refers to an integer variable; for example:

  
  on Exception do exceptionHandler(exception, currentObj, n);

  The **exception** object must be the first parameter that you specify when arming an exception handler.

**raise exception** Instruction

The **raise exception** instruction raises an exception.

**Syntax**

The syntax of the **raise exception** instruction is:

```plaintext
raise expression [internal | precondition];
```

**Description**

Specify an optional **internal** or **precondition** keyword in the **raise exception** instruction if you want to specify if the exception is an internal error or a precondition exception on the **raise exception** instruction.

**Note** As the **raise exception** instruction is implemented only for user exceptions, do not use it to raise **object locked** exceptions.
A **precondition** exception reports precondition contract exception where the caller has failed to satisfy precondition requirements of the invoked method. When a method raises a precondition exception, it signifies that its caller has failed to meet certain requirements such as passing valid parameters or ensuring a required state in objects used by the invoked method for the called method to do its job. When a **precondition** exception is raised at run time, the exception object contains two method descriptor references: `currentMethodDesc` describing the calling method and `reportingMethodDesc` describing the reporting method (that is, the method that raised the exception).

When an **internal** exception is raised, `currentMethodDesc` describes the method raising the exception and `reportingMethodDesc` contains a null reference. The default exception handler and dialog display information from the current method descriptor and the reporting method descriptor, if present.

**Example**

The following example shows a method used to raise an exception of type `MyException`.

```jade
raiseMyException(code: Integer; category: Integer);
vars
  exObj : MyException;
begin
  create exObj;
  exObj.errorCode := code;
  exObj.category := category;
  raise exObj;
end;
```

For more information about raising exceptions, see Chapter 3, "Exceptions".

**Transaction and Concurrency Control Instructions**

The JADE instructions for transaction and concurrency control are listed in the following table.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beginTransaction</code></td>
<td>Indicates the start of a transaction that updates persistent objects</td>
</tr>
<tr>
<td><code>beginTransientTransaction</code></td>
<td>Indicates the start of a transaction that updates shared transient objects</td>
</tr>
<tr>
<td><code>commitTransaction</code></td>
<td>Commits changes made within a transaction to storage</td>
</tr>
<tr>
<td><code>commitTransientTransaction</code></td>
<td>Releases all transaction duration shared transient locks and exits from transient transaction state</td>
</tr>
<tr>
<td><code>abortTransaction</code></td>
<td>Aborts the current transaction</td>
</tr>
<tr>
<td><code>abortTransientTransaction</code></td>
<td>Releases all transaction duration shared transient locks and exits from transient transaction state</td>
</tr>
<tr>
<td><code>beginLoad</code></td>
<td>Starts a read-only transaction</td>
</tr>
<tr>
<td><code>endLoad</code></td>
<td>Ends a read-only transaction started with the <code>beginLoad</code> instruction</td>
</tr>
<tr>
<td><code>beginLock</code></td>
<td>Starts a read-only transaction, ensuring referenced objects are the latest editions</td>
</tr>
<tr>
<td><code>endLock</code></td>
<td>Ends a read-only transaction started with the <code>beginLock</code> instruction</td>
</tr>
</tbody>
</table>
The `beginLoad` and `endLoad` instruction pair and the `beginLock` and `endLock` instruction pair perform similar functions. They enable you to defer the unlocking of locked objects to the end of the operations bounded by the `beginLoad` and `endLoad` instructions or the `beginLock` and `endLock` instructions. One `endLoad` or `endLock` instruction enables you to unlock all of the objects locked during a sequence of operations bounded by these instructions.

The difference between the two instruction pairs is that the `lock` pair automatically places a shared lock on all objects that are referenced, whereas the `load` pair leaves it to you to decide and code the objects that are locked. Therefore, if the object is already in cache and you do not place a shared lock on it or use some other means of forcing the latest copy into your local cache when using the `load` instruction pair, reading this object will use the copy in cache, which may be out of date.

In persistent transaction state, all unlock requests for persistent objects are ignored. Similarly, in transient transaction state, all unlock requests for shared transient objects are ignored. A session lock is therefore not released if the unlock request is made while in transaction state. To release a session lock, the unlock request must be made while not in transaction state.

For more details about object editions, see "Unlocking Objects", in Chapter 6. For details about reading and writing transactions, see "Using Read and Write Transactions", in the following subsection. See also "Sharing Uncommitted Persistent Objects", later in this section.

### Using Read and Write Transactions

The `abortTransaction` and `commitTransaction` instructions perform the following actions:

- Release all locks held at the time, including locks obtained in a server method that were not released after the server method execution.
- Finalize any existing load state (caused by a `beginLoad` instruction) or lock state (caused by a `beginLock` instruction).

The `abortTransaction` and `commitTransaction` instructions have priority over load state and lock state; that is, executing either instruction will release all transaction locks held by the process and finalize transaction state, load state, and lock state. For example, in the following code fragment, load state is ended by the `commitTransaction` instruction and the shared lock on `global` is released.

```java
// Initially, we're not in load state or transaction state, and // global is not locked
write "1: load state = " & process.isInLoadState.String; // false
write "1: tran state = " & process.isInTransactionState.String; // false
write "1: global lock = " & process.isLockedByMe(global).String; // false

// Begin load state. Manual unlocks of transaction duration locks // are ignored. All transaction duration locks will be released // when we exit load state (via either endLoad, commitTransaction, // or abortTransaction).
beginLoad;

// Place a shared lock on global ...
sharedLock(global);
// ... and then unlock it. This illustrates that the manual unlock // will be ignored, as we are in load state. The write statement // below will show that global is still locked.
unlock(global);

write "2: load state = " & process.isInLoadState.String; // true
write "2: tran state = " & process.isInTransactionState.String; // false
```

DevRef - 2018.0.01
write "2: global lock = " & process.isLockedByMe(global).String; // true
// Begin a transaction. This puts us in transaction state.
beginTransaction;

write "3: load state = " & process.isInLoadState.String; // true
write "3: tran state = " & process.isInTransactionState.String; // true
write "3: global lock = " & process.isLockedByMe(global).String; // true

// Commit the transaction. This ends transaction state and load
// state. All transaction duration locks are released, regardless
// of whether or not a manual unlock has been done. Even if the
// manual unlock of global was not done above, global would still
// be unlocked by the commitTransaction. An abortTransaction does
// the same, except that persistent object updates are not
// committed to the database. The write statement below will
// show that global has been unlocked.
commitTransaction;

write "4: load state = " & process.isInLoadState.String; // false
write "4: tran state = " & process.isInTransactionState.String; // false
write "4: global lock = " & process.isLockedByMe(global).String; // false

// End load state. This has already been done by the commitTransaction.
// Executing an endLoad when not in load state does nothing (it does
// not raise an exception).
endLoad;

write "5: load state = " & process.isInLoadState.String; // false
write "5: tran state = " & process.isInTransactionState.String; // false
write "5: global lock = " & process.isLockedByMe(global).String; // false

The abortTransaction instruction can be used, even when not in transaction state, to finalize load state and lock state and to release all transaction locks. For example:

// Initially, we're not in load state and global is not locked
write "1: load state = " & process.isInLoadState.String; // false
write "1: global lock = " & process.isLockedByMe(global).String; // false

// Begin load state. All transaction duration locks will be
// released when we exit load state (via either endLoad,
// commitTransaction or abortTransaction).
beginLoad;

// Place a shared lock on global
sharedLock(global);

write "2: load state = " & process.isInLoadState.String; // true
write "2: global lock = " & process.isLockedByMe(global).String; // true

// Execute an abortTransaction. Even though we're not in transaction
// state, this still ends load state and lock state, and releases all
// transaction duration locks currently held by the process.
abortTransaction;
write "3: load state = " & process.isInLoadState.String; // false
write "3: global lock = " & process.isLockedByMe(global).String; // false

Sharing Uncommitted Persistent Objects

JADE detects if the sharing of uncommitted persistent objects changes across processes and raises the appropriate exceptions should this occur. JADE enforces the starting, performing, and finalizing of persistent transactions on a single node (that is, a client or a server node). All of the update operations of the transaction must occur in the same node that started the transaction.

JADE does not attempt to synchronize the client cache and server cache when accessed by the same process. As a rule, you should not rely on changes made to persistent objects or a non-committed transaction being visible to server processes executed during that interval. (For details about object editions, see "Using Object Editions" under "Unlocking Objects", in Chapter 6. See also "Overview", in that chapter.)

beginTransaction Instruction

The beginTransaction instruction indicates the start of a transaction that updates persistent objects.

Syntax

The syntax of the beginTransaction instruction is:

beginTransaction;

Description

Persistent transactions allow updating of persistent objects only. The beginTransaction instruction causes the JADE Object Manager to place the process in transaction state. For more details, see "Locking Objects", in Chapter 6. Use the beginTransientTransaction instruction to update shared transient objects.

A persistent transaction has three effects, as follows.

- It allows persistent objects to be created, deleted, or updated. (An exception is raised if you attempt to create, delete, or update persistent objects outside of a transaction.)

- It coordinates the locking of persistent objects. Persistent objects are automatically locked (with an exclusive lock) as they are updated within a transaction. The locks on these objects are released at the next commitTransaction instruction. For details about reading and writing transactions, see "Using Read and Write Transactions", earlier in this chapter.

- It defers committing of updates to the database until the transaction is committed. Updates to persistent objects are not visible to other users until the next commitTransaction instruction is executed.

No transaction is necessary when creating, deleting, or updating non-shared objects. These objects can be updated at any time. (For details about sharing uncommitted persistent objects, see "Sharing Uncommitted Persistent Objects", earlier in this chapter.)

Executing the beginTransaction instruction is prohibited in the current process when the prohibited parameter in the Process class prohibitPersistentUpdates method is set to true.

For details about the restrictions that apply to transactions when using the serverExecution or clientExecution method option, see "Server and Client Execution Restrictions", earlier in this chapter.

Tip To avoid having to specify beginTransaction; in full, you can simply press the Ctrl+Shift+B keys at the position in the method at which the instruction is to be inserted.
Example

The following example shows the use of the `beginTransaction` instruction.

```java
initCompany();
begin
  company := Company.firstInstance;
  if company = null then
    beginTransaction;
    create company;
    company.name := "MegaCorp Inc";
    commitTransaction;
  endif;
end;
```

`beginTransaction` Instruction

The `beginTransaction` instruction indicates the start of a transaction that updates shared transient objects; that is, transient objects that are to be shared by two or more processes on the same node.

Syntax

The syntax of the `beginTransaction` instruction is:

```java
beginTransaction;
```

Description

Transient transactions allow the updating of shared transient objects only. (Use the `beginTransaction` instruction to update persistent objects.)

For details about creating shared transient objects, see "create Instruction", later in this chapter. See also "Object Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

The `beginTransaction` instruction causes the JADE Object Manager to place the process in transient transaction state. For more details, see "Locking Objects", in Chapter 6.

A transient transaction has two effects, as follows.

- It allows shared transient objects to be created, deleted, or updated. (An exception is raised if you attempt to create, delete, or update shared transient objects outside of a transient transaction.)

- It coordinates the locking of shared transient objects.

  Shared transient objects are automatically locked as they are updated within a transient transaction. The locks on these objects are released at the next `commitTransientTransaction` instruction.

Use the `sharedTransient` modifier of the `create` instruction to create transient objects, which can be shared across threads; for example:

```java
create obj sharedTransient;
```

Shared transient objects can be updated only within transient transaction state. All locks (both manual and automatic) of shared transient objects are released at the end of the transient transaction. The transient transaction state is specified by the `beginTransactionTransaction` and `commitTransientTransaction` instructions.

You should explicitly create transient objects that need to be shared across processes in a node as `shared` transient objects, which are updated only within transient transactions.
Updates to shared transient objects are applied when the `commitTransientTransaction` instruction is executed. As shared transient objects are exclusively locked while being updated, other processes cannot lock these objects until the `commitTransientTransaction` instruction is executed. Other processes attempting to access these objects without locking will not see the uncommitted updates but instead will view the most-recently committed editions.

No transaction is necessary when creating, deleting, or updating ordinary non-shared transient objects. These objects can be updated at any time.

All lock operations are enforced for shared transient objects. These actions guarantee an access protocol for shared transient objects similar to that of persistent objects, which is essential for the execution of synchronized multithreaded update operations. For more details about transient objects, see:

- "Class Class" and "Object Class", in Chapter 1 of the JADE Encyclopaedia of Classes
- "JADE Object Handling", in Chapter 1 of the JADE Object Manager Guide

Example

The following example shows the use of the `beginTransientTransaction` instruction.

```java
createArray(anObj: Object) updating;
vars
  array : ObjectArray;
begin
  beginTransientTransaction;
  create arr sharedTransient;
  array.add(anObj);
  commitTransientTransaction;
end;
```

`commitTransaction` Instruction

The `commitTransaction` instruction commits changes made within a transaction to persistent storage.

Syntax

The syntax of the `commitTransaction` instruction is:

```java
commitTransaction;
```

Description

Persistent transactions update persistent objects only. (Use the `commitTransientTransaction` instruction to update transient objects.)

The `commitTransaction` instruction requests the JADE Object Manager to make permanent the changes that were made to persistent objects after a `beginTransaction` instruction; that is, to commit them to physical storage on a database server. It also causes locks to be released for objects that were updated within the transaction.

**Tip** To avoid having to specify `commitTransaction` in full, you can simply press the Ctrl+Shift+C keys at the position in the method at which the instruction is to be inserted.

Executing the `commitTransaction` instruction is prohibited in the current process when the `prohibited` parameter in the `Process` class `prohibitPersistentUpdates` method is set to `true`.

For details about reading and writing transactions, see "Using Read and Write Transactions", earlier in this chapter and for details about the restrictions that apply to transactions when using the `serverExecution` or `clientExecution` method option, see "Server and Client Execution Restrictions", earlier in this chapter.
Example

The following example shows the use of the `commitTransaction` instruction in a method on `Employee` class that is used to update employee details.

```java
updateDetails(address: String; phone: String; dateOfBirth: Date) updating;
begin
  beginTransaction;
  homeAddress := address;
  homePhone := phone;
  birthDate := dateOfBirth;
  commitTransaction;
end;
```

**commitTransientTransaction Instruction**

The `commitTransientTransaction` instruction releases all transaction duration shared transient locks and exit transient transaction state.

**Syntax**

The syntax of the `commitTransientTransaction` instruction is:

```sql
commitTransientTransaction;
```

**Description**

Transient transactions apply only to shared transient objects. (Use the `commitTransaction` instruction to update persistent objects.)

For details about creating shared transient objects, see "create Instruction", later in this chapter. See also "Object Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

The `commitTransientTransaction` instruction commits updates to shared transient objects and releases the updated objects so that other processes can lock them and view the updates.

For details about the restrictions that apply to transactions when using the `serverExecution` or `clientExecution` method option, see "Server and Client Execution Restrictions", earlier in this chapter.

**Example**

The following example shows the use of the `commitTransientTransaction` instruction.

```java
deleteArray(theSharedArray: ObjectArray);
begin
  beginTransientTransaction;
  delete theSharedArray;
  commitTransaction;
end;
```

**abortTransaction Instruction**

The `abortTransaction` instruction aborts the current transaction.
Syntax

The syntax of the `abortTransaction` instruction is:

```plaintext
abortTransaction;
```

Description

The `abortTransaction` instruction requests the JADE Object Manager to undo any changes made to persistent objects since the previous `beginTransaction`. It also causes locks to be released for objects that were updated within the transaction.

If you use this instruction when not in transaction state, all transaction duration locks on persistent objects will be released.

For details about reading and writing transactions, see "Using Read and Write Transactions", earlier in this chapter.

Example

The following example shows the use of the `abortTransaction` instruction.

```plaintext
bAdd_click(button: Button input) updating;
vars
  err : String;
begin
  beginTransaction;
  create customer;
  err := customer.loadSelf(company,
                           customerName.text,
                           address.text,
                           contact.text);
  if err <> "" then
    abortTransaction; // Never put message boxes up
    app.msgBox(err, "Error", 0); // while in transaction state.
    customer := null;
    return;
  endif;
  commitTransaction;
  unloadForm;
end;
```

`abortTransientTransaction` Instruction

The `abortTransientTransaction` instruction undoes any changes made to shared transient objects since the previous `beginTransaction` instruction and exits from transient transaction state. In addition, all transaction duration shared transient locks are released.

Syntax

The syntax of the `abortTransientTransaction` instruction is:

```plaintext
abortTransientTransaction;
```

Description

Transient transactions apply only to shared transient objects. (Use the `abortTransaction` instruction to abort the current transaction of a persistent object.)
If you use this instruction when not in transient transaction state, all transaction duration locks on shared transient objects will be released. For details about creating shared transient objects, see "create instruction", later in this chapter. See also "Object Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

**Example**

The following example shows the use of the `abortTransientTransaction` instruction.

```java
deleteArray(theSharedArray: ObjectArray);
begin
    beginTransientTransaction;
    delete theSharedArray;
    abortTransientTransaction;
end;
```

### `beginLoad` and `endLoad` Instructions

The `beginLoad` and `endLoad` instructions bracket an atomic read-only transaction.

**Syntax**

The syntax of the `beginLoad` instruction and the `endLoad` instruction is:

```java
beginLoad;
endLoad;
```

**Description**

All database objects locked between the current `beginLoad` and `endLoad` instructions remain locked until the `endLoad` instruction is executed at which time they are unlocked. Objects locked with session duration and objects locked before the current `beginLoad` instruction are not unlocked by the `endLoad` instruction.

Unlock object requests are ignored between `beginLoad` and `endLoad` instructions.

Read transactions can be nested up to a maximum level of 255; that is, a `beginLoad` and `endLoad` transaction can be nested inside another `beginLoad` and `endLoad` transaction or inside a `beginLock` and `endLock` transaction.

---

**Note**  
`commitTransaction` and `abortTransaction` instructions end any current read-only transactions and release all transaction duration locks.

An object must be locked directly or indirectly to guarantee that the latest edition of the object is loaded. (For more details about object editions, see "Unlocking Objects", in Chapter 6.)

For details about reading and writing transactions, see "Using Read and Write Transactions", earlier in this chapter, and for details about the restrictions that apply to transactions when using the `serverExecution` or `clientExecution` method option, see "Server and Client Execution Restrictions", earlier in this chapter.

**Example**

The following example shows the use of the `beginLoad` and `endLoad` instructions.

```java
getBalance(client: Client): Decimal;
vars
    balance : Decimal[19,2];
begin
    balance := 0;
    beginLoad;
    foreach account in client.accounts do
```
sharedLock(account);
    balance := balance + account.getBalance;
endforeach;
// unlock all accounts
endLoad;
return balance;
end;

In this example, a shared (read) lock is acquired on each account as it is processed. The shared lock ensures that the account objects remain consistent for the duration of the totaling of account balances.

The foreach instruction implicitly acquires a shared lock on the client.accounts collection, which prevents accounts being added or deleted for the duration of the processing loop. This ensures that the totaling process sees a consistent view of accounts.

**beginLock and endLock Instructions**

The beginLock and endLock instructions bracket a read-only transaction to ensure that objects referenced after executing the current beginLock instruction and before the endLock instruction are the latest editions of the objects.

**Syntax**

The syntax of the beginLock instruction and the endLock instruction is:

```
beginLock;
endLock;
```

**Description**

As objects are referenced, an implicit shared lock is acquired on each object, which causes the latest edition to be fetched from the server, if required. (For more details about object editions, see "Unlocking Objects", in Chapter 6.)

Read transactions can be nested up to a maximum level of 255; that is, a beginLock and endLock transaction can be nested inside another beginLock and endLock transaction or inside a beginLoad and endLoad transaction.

For details about reading and writing transactions, see "Using Read and Write Transactions", earlier in this chapter and for details about the restrictions that apply to transactions when using the serverExecution or clientExecution method option, see "Server and Client Execution Restrictions", earlier in this chapter.

**Example**

The following example shows the use of the beginLock and endLock instructions to acquire an implicit shared lock on each car object as it is referenced.

```
vars
totalAge : Integer;
allCars : CarsArray;
car   : Car;
begin
    create allCars transient;
    Car.allInstances(allCars, 0, Car.abstract);
    beginLock;
        foreach car in allCars do
            totalAge := totalAge + car.calculateAge;
        endforeach;
    endLock;
write "Average age: " & (totalAge/allCars.size).String;
```
delete allCars;
end;

Instructions for Creating and Deleting Objects

The JADE instructions for creating and deleting objects are listed in the following table.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>create</td>
<td>Creates an instance of a class, and assigns it to a variable or a property</td>
</tr>
<tr>
<td>delete</td>
<td>Deletes an object</td>
</tr>
</tbody>
</table>

Your applications can create or delete persistent instances only for the `RootSchema` classes listed in the following table. If you attempt to create or delete a persistent instance of any other `RootSchema` class, an exception is raised.

<table>
<thead>
<tr>
<th>BinaryArray</th>
<th>BooleanArray</th>
<th>CharacterArray</th>
<th>DateArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>DecimalArray</td>
<td>HugeStringArray</td>
<td>IntegerArray</td>
<td>ObjectArray</td>
</tr>
<tr>
<td>ObjectLongNameDict</td>
<td>ObjectSet</td>
<td>OleObject</td>
<td>PointArray</td>
</tr>
<tr>
<td>RealArray</td>
<td>Rectangle</td>
<td>RectangleArray</td>
<td>SortActor</td>
</tr>
<tr>
<td>SortActorArray</td>
<td>Sound</td>
<td>StringArray</td>
<td>TimeArray</td>
</tr>
<tr>
<td>TimeStampArray</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can use class lifetime options to restrict the creation of objects. For details, see "Class Lifetime Options", in Chapter 1 of the JADE Development Environment User's Guide.

create Instruction

The `create` instruction creates an instance of a class, and assigns it to a variable or a property.

Syntax

The syntax of the `create` instruction is:

```
create entity [as expression] [transient | persistent | sharedTransient];
```

The `entity` value can be a property expression, a local variable, or a formal parameter.

The syntax of the extended `create` instruction, which retrieves the object reference using an assignment instruction, is as follows.

```
reference := create class-name(parameters) lifetime;
```

An assignment statement is required. You cannot use this syntax in a general assignment expression; for example, as a value passed to a parameter in a method call or the value in a return statement.

Description

You can optionally specify a class expression in the `as` clause of the `create` instruction but you cannot specify a class expression in the `as` clause of the extended `create` instruction. The class does not need to be an explicit class name; it can be a variable or any valid expression that yields a class. However, the class name must be explicit in the extended `create` instruction.
The as clause in the extended create instruction is not supported, because the class name must be specified so that the compiler can validate the signature of the create method call.

You cannot use the create instruction without parameters if the class of the object being created has a constructor that has parameters.

The extended create instruction syntax is allowed for any class, with or without an explicit constructor with or without parameters.

The transient, persistent, and sharedTransient modifiers specify whether the instance of the class being created is transient, shared transient, or persistent. If the modifier is omitted, the default lifetime for the class is used.

For details about:
- Constructor create methods, see "Using Constructors and Destructors", in Chapter 4 of the JADE Development Environment User's Guide.
- Shared transient transactions, see "beginTransientTransaction", earlier in this chapter. See also "Sharing Uncommitted Persistent Objects".
- For details about the RootSchema classes for which you can create persistent instances, see "Instructions for Creating and Deleting Objects".

The create instruction on a Form subclass creates a GUI form. If you want to create a print form at run time that simulates the entire GUI process, use the GUIClass class createPrintForm method. (For details, see Chapter 1 of the JADE Encyclopaedia of Classes.)

create Instruction Examples

The following example shows the use of the create instruction without parameters.

```jade
createEmployee(newName, newAddress, newPhone: String;
    newBirthDate: Date): Employee;
vars
    newEmployee : Employee;
begin
    create newEmployee;
    newEmployee.name := newName;
    newEmployee.address := newAddress;
    newEmployee.phone := newPhone;
    newEmployee.birthDate := newBirthDate;
return newEmployee;
end;
```

The following example shows the use of the as clause in the create instruction.

```jade
createFault(faultType: Character): Fault;
vars
    fault : Fault;
begin
    if faultType = "N" then
        create fault as NewFeatureSuggestion;
    elseif faultType = "D" then
        create fault as DocumentationFault;
    else
        create fault as NormalFault;
    endif;
... // Do other initialization
```
return fault;
end;

The following example shows the use of the create instruction without parameters when the class is not an explicit class name.

createFault(faultClass: Class): Fault;
vars
  fault : Fault;
begin
  create fault as faultClass;
  ...  // Do other initialization
  return fault;
end;

Extended create Instruction Examples

The following example is an Employee class create method with parameters.

create(newName, newAddress, newPhone: String; newBirthDate: Date) updating;
begin
  name := newName;
  address := newAddress;
  phone := newPhone;
  birthDate := newBirthDate;
end;

The following is an example of an extended create instruction based on the above Employee class create method with parameters example.

createEmployee(newName, newAddress, newPhone: String;
  newBirthDate: Date): Employee;
vars
  newEmployee : Employee;
begin
  newEmployee := create Employee(newName, newAddress, newPhone, newBirthDate);
  return newEmployee;
end;

delete Instruction

The delete instruction deletes an object.

Syntax

The syntax of the delete instruction is:

  delete expression;

The expression value evaluates to an object reference.

Description

The action of the delete instruction is first to evaluate the expression and then to attempt to delete the object that the expression references.

If the referenced object exists, it is deleted. If the object does not exist, an exception is raised, but no exception is raised if the reference is null.
The value of a deleted local variable or property is set to null by the delete instruction, so that the deleted object cannot be inadvertently accessed at some later stage.

The operand of a delete instruction must be capable of being assigned to. For example, an error is reported if you attempted to delete a usage input parameter. The parameter would have to be made usage io or output for the code to compile without error.

For details about the RootSchema classes for which you can delete persistent instances, see "Instructions for Creating and Deleting Objects", earlier in this chapter.

**Example**

The following example uses the delete instruction to clear an array.

```jade
removeGraph(graphName: String) updating;
vars
    graph : Object;
    count : Integer;
begin
    // because the graph has been stopped, we must remove all
    // data that is used to create the graph
    count := 1;
    while count <= myLines.size do
        graph := myLines[count];
        if graph.IGLines.name = graphName then
            self.myLines.removeAt(count);
            delete graph.IGLines;
            count := count - 1;
        endif;
        count := count + 1;
    endwhile;
end;
```

**JADE Language Vocabulary**

This section contains:

- Special Symbols used in the JADE language
- JADE Reserved Words
**Special Symbols**

The special symbols used in the JADE language are listed in the following table.

<table>
<thead>
<tr>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>^</th>
<th>@</th>
<th>&amp;</th>
<th>(</th>
</tr>
</thead>
<tbody>
<tr>
<td>)</td>
<td>[</td>
<td>]</td>
<td>=</td>
<td>&lt;</td>
<td>&gt;</td>
<td>&lt;=</td>
<td>&gt;=</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>:</td>
<td>.</td>
<td>,</td>
<td>;</td>
<td>:=</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For details about using these symbols as operators when constructing composite expressions, see "Expressions with Operators", earlier in this chapter.

**Reserved Words**

When you are creating a JADE application, you may find that some words cannot be used in your JADE methods (or you can use them in certain ways only). These are *reserved words*. Reserved words have a special meaning in JADE; for example, if you try to create a method called `return`, you are informed that `return` is a reserved word.

The JADE reserved words are listed in the following table.

<table>
<thead>
<tr>
<th>abortTransaction</th>
<th>abortTransientTransaction</th>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td>app</td>
<td>appContext</td>
<td></td>
</tr>
<tr>
<td>as</td>
<td>attributeDefinitions</td>
<td></td>
</tr>
<tr>
<td>beginLoad</td>
<td>beginLock</td>
<td></td>
</tr>
<tr>
<td>beginTransientTransaction</td>
<td>beginTransaction</td>
<td></td>
</tr>
<tr>
<td>break</td>
<td>Binary</td>
<td></td>
</tr>
<tr>
<td>Boolean</td>
<td>call</td>
<td></td>
</tr>
<tr>
<td>break</td>
<td>Character</td>
<td></td>
</tr>
<tr>
<td>classMapDefinitions</td>
<td>commitTransaction</td>
<td></td>
</tr>
<tr>
<td>continue</td>
<td>continue</td>
<td></td>
</tr>
<tr>
<td>currentSchema</td>
<td>currentSession</td>
<td></td>
</tr>
<tr>
<td>defaultFileDefinition</td>
<td>databaseDefinitions</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td>div</td>
<td></td>
</tr>
<tr>
<td>documentationText</td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>_encryptedSource</td>
<td>_endEncryptedSource</td>
<td></td>
</tr>
<tr>
<td>endforeach</td>
<td>end</td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td>endLock</td>
<td></td>
</tr>
<tr>
<td>endwhile</td>
<td>epilog</td>
<td></td>
</tr>
<tr>
<td>eventMethodMappings</td>
<td>exportedClassDefinitions</td>
<td></td>
</tr>
<tr>
<td>exportedConstantDefinitions</td>
<td>exportedInterfaceDefinitions</td>
<td></td>
</tr>
<tr>
<td>exportedMethodDefinitions</td>
<td>exportedPropertyDefinitions</td>
<td></td>
</tr>
<tr>
<td>_exposedConstantDefinitions</td>
<td>_exposedListDefinitions</td>
<td></td>
</tr>
<tr>
<td>_exposedMethodDefinitions</td>
<td>_exposedJavaFeatures</td>
<td></td>
</tr>
</tbody>
</table>
Words that begin with double underscore characters (__) are reserved for internal JADE use. In addition, the first four characters of a user-defined name should not be Jade (which is used to prefix JADE system entities).

The following words are not reserved words, but are part of the JADE syntax. Although you can reuse these words, the JADE editor applies the editor preferences color. To avoid confusion, you should not use these words as user-defined values.

- **discreteLock** (specifies that a shared lock is not retained for the duration of the foreach instruction)
- **inheritCreate** (instruction that invokes the superclass constructor with parameters)
- **inheritMethod** (instruction that invokes the superclass implementation of a method)
- **internal, precondition** (optional keywords specifying the type of exception in the `raise` instruction)
- **persistent, transient, sharedTransient** (optional modifiers for the `create` instruction)
- **setPatchVersion** (schema definition)
Chapter 2  Notifications

This chapter covers the following topics.

- **Overview**
  - Using Notifications
  - System Notifications
  - User Notifications
- **Beginning a Notification**
  - Example
  - Receiving a Notification
    - Receiving System Notifications
    - Receiving User Notifications
- **Ending a Notification**
- **Using Events**
  - Causing a Class Event
  - Causing an Object Event

**Overview**

Notifications communicate specified events, or actions, to the user application. Notifications ensure that concurrent access to data does not affect system integrity, and provide formalized communication within an application where control has been dispersed and data is constantly changing.

A notification serves as an automatic notice board, where a method can post details of an event so that any interested party can be informed automatically. The informing method does not need to know who is to be informed, and those who are informed do not need to know who caused the event.

Notifications containing binary and string (Binary, String, StringUtf8) data of up to 48K bytes can be sent across the network. For applications running within the server node, the limit for notifications containing binary or string data is 2G bytes. Note, however, that this applies only to single user and server applications. In multiuser applications, persistent notifications are sent via the database server, even if the receiving process is on the same node as the sender. In notification cause events, exception 1267 (Notification info object too big) is raised if the binary or string userInfo data exceeds the applicable value.

JADE notifications may have a differing execution order when intermixed with window events in JADE thin client mode. This difference arises because the notifications occur on the application server rather than the presentation client. Notifications are usually interlaced with any window events that may occur.

In JADE thin client mode, the notification occurs when the application server thread processing the presentation client operations becomes idle. However, the presentation client may also be idle and send event notifications such as form activations, focus changes, and so on, at the same time. This asynchronous operation may result in a slightly different execution order for these events from that experienced in JADE when it is not running in thin client mode.
Caution If the application will be run in JADE thin client mode, subscribe to notifications and timers with care. When an event occurs, each registered subscriber is notified on the application server. In thin client mode, the first notification or timer that is sent causes a further message to be sent to the presentation client to post a message into the Windows message queue of the client. (This is necessary so that the subsequent execution of the notification logic is synchronized with what is taking place on the presentation client.)

When that Windows message is processed by the presentation client, another message is sent to the application server to initiate the processing of the first 10 queued notifications or timers for that client. If there are more than 10 notifications, these actions are repeated until all queued events are processed.

Notifications and timers could therefore have a considerable impact on network traffic.

Ensure that the data being passed on a notification is less than the recommended size limit. In addition, avoid sending large amounts of data using the notification. It is better to pass an object reference, rather than a large primitive type value.

For a discussion about class notifications, object notifications, and registering notifications, see the Erewhon.pdf file that is included in the erewhon subdirectory of the examples directory on the JADE release medium.

Using Notifications

JADE provides system and user notifications, to enable you to notify a specified object when an event has occurred.

You can register a request to be notified of events that occur in any instance of a specific class (by using the Object class `beginClassNotification` or `beginClassesNotification` method) or in a specific instance of a class (by using the Object class `beginNotification` method). In addition, you can register a request to be notified of events that occur in a specified interface whose event methods are implemented in any instance of a class (by using the Object class `beginClassNotificationForIF`, `beginClassesNotificationForIF`, or `beginNotificationForIF` method).

Notes A process that uses an Object class `beginClassNotification` or `beginClassesNotification` method to subscribe to user event notifications for transient instances will receive notifications for all shared transient instances and for those non-shared transient instances that it has created (that is, the process will not receive or cause notifications for non-shared transient instances that have been created by other processes).

You can use the Object class `beginClassNotificationForIF`, `beginClassesNotificationForIF`, or `beginNotificationForIF` method to register a request to be notified of events that occur in a specified interface whose event methods are implemented in a class instance.

Non-immediate events caused on transient objects are not discarded when a persistent transaction is aborted. For example, if the receiver of a causeEvent is a shared transient instance, any notifications are held when the transaction is aborted and delivered when the next transaction commits.

The Notification class provides the properties summarized in the following table, which enable you to determine if an interface subscribed to a notification.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>featureNumber</td>
<td>Contains a value that allows for identification of the interface method that was mapped by the subscriber</td>
</tr>
<tr>
<td>isInterface</td>
<td>Specifies whether the notification was registered by an interface notification method</td>
</tr>
<tr>
<td>typeName</td>
<td>Contains a value that allows for identification of the associated interface</td>
</tr>
</tbody>
</table>
System Notifications

System notifications are triggered by database activities such as creating a new object, updating an object, or deleting an object. As the JADE Object Manager processes these activities, it issues system notifications to any object that has registered an interest in the event.

Note System notifications are invoked for persistent objects only.

User Notifications

User notifications enable you to send application-specific events that are outside the scope of the events for which system notifications are sent. For example, a form in an application may want to send a notification in response to a button being clicked.

To cause a user notification, a method uses the Object class causeEvent, sdeCauseEvent, or sdsCauseEvent method to inform the JADE Object Manager that the event has occurred. The JADE Object Manager then sends the notification to all objects that have registered an interest in it. For details about Synchronized Database Service (SDS) events and notifications, see “Persistent Events and Notifications”, in Chapter 10.

Beginning a Notification

Use the Object class beginClassNotification method to register the receiver to be notified when a nominated event occurs on instances of a class and its subclasses. An object that subscribes to a class notification is notified when the nominated event occurs for any instance of the specified class or its subclasses. The signature of the beginClassNotification method is:

```java
beginClassNotification(theClass: Class; 
                        transients: Boolean; 
                        eventType: Integer; 
                        responseType: Integer; 
                        eventTag: Integer);
```

Use the Object class beginClassesNotification method to register the receiver to be notified when a nominated event occurs on instances of a class and optionally its subclasses, without any additional searches for subschema copies in the current schema. An object that subscribes to a class notification is notified when the nominated event occurs for any instance of the specified class or its subclasses. The signature of the beginClassesNotification method is:

```java
beginClassesNotification(theClass: Class; 
                         includeSubclasses: Boolean; 
                         transients: Boolean; 
                         eventType: Integer; 
                         responseType: Integer; 
                         eventTag: Integer);
```

Use the Object class beginNotification method to register the receiver to be notified when a nominated event occurs on a specified object. An object that subscribes to an object notification is notified when the specified event occurs for the notification target. The signature of the beginNotification method is:

```java
beginNotification(theObj: Object; 
                  eventType: Integer; 
                  responseType: Integer; 
                  eventTag: Integer);
```
Use the `Object` class `beginClassNotificationForIF, beginClassesNotificationForIF,` and `beginNotificationForIF` method variations to register a request to be notified of events that occur in a specified interface whose event methods are implemented in a class instance.

The parameters for methods that begin notifications are listed in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventTag</td>
<td>Identifies a specific notification when you have multiple subscriptions for the same event type.</td>
</tr>
<tr>
<td>eventType</td>
<td>Specifies the type of event for which the notification is registered.</td>
</tr>
<tr>
<td>responseType</td>
<td>Specifies the frequency with which the event is notified.</td>
</tr>
<tr>
<td>theClass</td>
<td>For the <code>beginClassNotification, beginClassesNotification, beginClassNotificationForIF,</code> and <code>beginClassesNotificationForIF</code> methods, specifies the class for which the notification is to be registered.</td>
</tr>
<tr>
<td>includeSubclasses</td>
<td>For the <code>beginClassesNotification</code> and <code>beginClassesNotificationForIF</code> methods, specifies whether subclasses are included or excluded in the notification registration</td>
</tr>
<tr>
<td>theObj</td>
<td>For the <code>beginNotification</code> and <code>beginNotificationForIF</code> methods, specifies the object for which the notification is to be registered. Note that if this object is transient, only user notifications can be received.</td>
</tr>
<tr>
<td>transients</td>
<td>For the <code>beginClassNotification, beginClassesNotification, beginClassNotificationForIF,</code> and <code>beginClassesNotificationForIF</code> methods, specifies if the user notification is to be registered for events that occur to transient instances (true), or persistent instances (false) of the class. You can subscribe to system notifications only for persistent objects; that is, the transient parameter must be false.</td>
</tr>
<tr>
<td>theInterface</td>
<td>The interface that defines the appropriate <code>userNotification</code> or <code>sysNotification</code> method implemented by the class instance.</td>
</tr>
</tbody>
</table>

Global constants in the `NotificationResponses` category, listed in the following table, provide the valid values for the `responseType` parameter.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response_Cancel</td>
<td>1</td>
<td>Sends a notification when the target notification object receives a matching event and then cancels the notification</td>
</tr>
<tr>
<td>Response_Continuous</td>
<td>0</td>
<td>Sends a notification whenever the target notification object receives a matching event</td>
</tr>
<tr>
<td>Response_Suspend</td>
<td>2</td>
<td>Sends a notification when the target notification object receives a matching event and then suspends notification until users refresh their local copy of the target object from the database</td>
</tr>
</tbody>
</table>

Global constants in the `SystemEvents` category provide the valid values, listed in the following table, for system event types that can be subscribed to in the `eventType` parameter.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any_System_Event</td>
<td>0</td>
<td>Object has been created, deleted, or updated</td>
</tr>
<tr>
<td>Object_Update_Event</td>
<td>3</td>
<td>Object has been updated</td>
</tr>
</tbody>
</table>
Global Constant | Integer Value | Description
--- | --- | ---
Object_Create_Event | 4 | Object has been created
Object_Delete_Event | 6 | Object has been deleted

Event types in the range 16 through `Max_Integer`, or `#7FFFFFFF` (2,147,483,647), are user-definable events. User notifications are sent for any event with an event type in this range.

Global constants in the `UserEvents` category provide the valid values, listed in the following table, for user event types that can be subscribed to in the `eventType` parameter are.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any_User_Event</td>
<td>-1 (to subscribe to all user events)</td>
</tr>
<tr>
<td>User_Base_Event</td>
<td>16</td>
</tr>
<tr>
<td>User_Max_Event</td>
<td><code>Max_Integer</code> (#7FFFFFFF, equating to 2147483647)</td>
</tr>
</tbody>
</table>

The `beginClassesNotification` and `beginClassesNotificationForIF` methods make no attempt to re-interpret the value of the `theClass` parameter, so that the calls do not look for a subschema copy class in the current schema with which to register the call and they optionally allow only the class without any of its subclasses to be registered for the notification.

For example, a `beginClassNotification(MemberKeyDictionary, false, Any_System_Event, Response_Continuous, 1)` call looks for a subschema copy class in the current schema to register and a `beginClassesNotification(MemberKeyDictionary, false, false, Any_System_Event, Response_Continuous, 1)` call registers the root `MemberKeyDictionary` class in the `RootSchema`.

If you want to specify your `MemberKeyDictionary` subschema copy class and allow both the class and its subclasses to be registered for the notification, call `beginClassesNotification` as follows.

```java
beginClassesNotification(currentSchema.getClass('MemberKeyDictionary'), true, false, Any_System_Event, Response_Continuous, 1);
```

The following table lists the methods that begin and end matching notification types.

<table>
<thead>
<tr>
<th>Method that Registers a Request for an Event Notification</th>
<th>Method that Terminates the Notification</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beginClassNotification</code></td>
<td><code>endClassNotification</code></td>
</tr>
<tr>
<td><code>beginClassesNotification</code></td>
<td><code>endClassesNotification</code></td>
</tr>
<tr>
<td><code>beginNotification</code></td>
<td><code>endNotification</code></td>
</tr>
<tr>
<td><code>beginClassNotificationForIF</code></td>
<td><code>endClassNotificationForIF</code></td>
</tr>
<tr>
<td><code>beginClassesNotificationForIF</code></td>
<td><code>endClassesNotificationForIF</code></td>
</tr>
<tr>
<td><code>beginNotificationForIF</code></td>
<td><code>endNotificationForIF</code></td>
</tr>
</tbody>
</table>

Alternatively, you can use the `endNotificationForSubscriber` method to terminate all notifications for a specified subscriber.

For more details, see the appropriate class under "Object Class", in Chapter 1 of the JADE Encyclopaedia of Classes. See also "Example", in the following section.
Example

The following example, from the Erewhon Investments example schema supplied on the JADE release medium, shows the use of the `beginNotification` method when retrieving the next subset of items in reverse order from a collection to fill a list box.

```plaintext
zLoadSubsetReversed(whichEnd: Integer) updating, protected
// The whichEnd parameter is used to fill the list from the end of the list
// box if EndOfList, otherwise fill starting from the top of the list box.
vars
    linesInDisplay, idx : Integer;
    obj : Object;
begin
    linesInDisplay := self.lines;
    idx := self.listCount;
    if whichEnd = EndOfList then
        // Fill from the end of the list box
        while zMyIterator.back(obj) do
            idx := idx + 1;
            // Add the item text and object
            addItem(displayEntry(self, obj, idx));
            itemObject[idx] := obj;
            // If the entry was selected, then select it in the list
            if zSelectedObjects.includes(obj) then
                self.itemSelected[idx] := true;
            endif;
            zCurrentIteratorPosition := idx;
            if showUpdates then
                // We want to be told about changes to this object
                beginNotification(obj, Object_Update_Event,
                                  Response_Continuous, NotifyInstanceUpdate);
            endif;
            if idx = linesInDisplay then
                break; // We've filled enough
            endif;
        endwhile;
    else
        // Fill from the start of the list box
        idx := 1;
        while zMyIterator.back(obj) do
            // Add the item text and object
            addItemAt(displayEntry(self, obj, idx), idx);
            itemObject[idx] := obj;
            // If the entry was selected, then select it in the list
            if zSelectedObjects.includes(obj) then
                itemSelected[idx] := true;
            endif;
            zCurrentIteratorPosition := idx;
            if showUpdates then
                // We want to be told about changes to this object
                beginNotification(obj, Object_Update_Event,
                                  Response_Continuous, NotifyInstanceUpdate);
            endif;
            if self.listCount = linesInDisplay then
                break; // We've filled enough
            endif;
        endwhile;
    endif;
```
idx := idx + 1;
endwhile;
endif;
end;

Receiving a Notification

This section covers the receiving of system and user notifications.

Receiving System Notifications

A system notification occurs when an event is triggered with an event type less than User_Base_Event (16). Non-GUI objects (that is, objects that are not instances of a Window subclass) respond to system notifications by implementing the Object class sysNotification method.

GUI objects (that is, objects that are instances of a Window subclass) respond to system notifications by implementing their Window subclass sysNotify event method.

The sysNotification method has the following signature.

sysNotification(eventType: Integer;
                 theObject: Object;
                 eventTag: Integer) updating, abstract;

The sysNotify method has the following signature.

sysNotify([control: Control-type input;] // parameter only for controls
          eventType: Integer;
          theObject: Object;
          eventTag: Integer) updating;

For examples of the sysNotification and sysNotify methods, see "System Notification Examples", later in this section.

eventType

The eventType parameter specifies the event for which the notification has been received.

The global constants (provided by the SystemEvents category) for the types of system event that can be received are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Object has been ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_Create_Event</td>
<td>4</td>
<td>Created</td>
</tr>
<tr>
<td>Object_Delete_Event</td>
<td>6</td>
<td>Deleted</td>
</tr>
<tr>
<td>Object_Update_Event</td>
<td>3</td>
<td>Updated</td>
</tr>
</tbody>
</table>

theObject

The theObject parameter specifies the object for which the event occurred.

Caution Attempts to access features of the object of a notification of a delete event (Object_Delete_Event) raise an exception.
Before a system notification is delivered, the system automatically clears the buffer of the target object (referenced by the `theObject` parameter) if one exists in the current node cache. This enables the attributes and references of an object that is referenced to reflect the update notified by the system notification.

**eventTag**

The `eventTag` parameter is a user-defined integer value (for example, an index into an array) that is used to identify a notification subscription that will be passed to the notification callback method.

**System Notification Examples**

The following example uses the `sysNotification` method to handle system notifications subscribed to by application objects.

```java
sysNotification(eventType: Integer; theObject: Object; eventTag: Integer)
    updating;
    begin
        if eventType = Object_Create_Event then       // new product
            self.loadTable;
        endif;
    end;
end;
```

The following example, from the Erewhon Investments example schema supplied on the JADE release medium, shows a `sysNotify` method that registers the current form to be notified when the nominated event (specified in the `eventType` parameter) occurs on the requested object.

```java
sysNotify(eventType: Integer; theObject: Object; eventTag: Integer)
    updating;
    begin
        app.mousePointer := Busy;
        zSynchronizeForm(eventType, theObject, eventTag, null);
        epilog
            app.mousePointer := Idle;
    end;
end;
```

**Receiving User Notifications**

A user notification occurs when an event is triggered with an event type that is greater than or equal to `User_Base_Event` (16).

Non-GUI objects (that is, objects that are not instances of a `Window` subclass) respond to user notifications by implementing the `Object` class `userNotification` method.

GUI objects (that is, objects that are instances of a `Window` subclass) respond to user notifications by implementing their `Window` subclass `userNotify` event method.

The `userNotification` method has the following signature.

```java
userNotification(eventType: Integer; theObject: Object; eventTag: Integer; userInfo: Any) updating;
```

The `userNotify` method has the following signature.

```java
userNotify([control: Control-type input]); // parameter only for controls
    eventType: Integer;
```
An exception is raised if the notification subscriber does not exist. To terminate the delivery of notifications to a subscriber, use the Object class endNotificationForSubscriber method to specify the subscriber whose previously registered Object class beginNotification, beginClassNotification, beginClassesNotification, beginClassNotificationForIF, beginClassesNotificationForIF, and beginNotificationForIF methods are to be terminated. (For details about terminating the delivery of notifications, see "Ending a Notification", later in this chapter.)

For examples of the userNotification and userNotify methods, see "User Notification Examples", later in this section.

**eventType**

The eventType parameter specifies the type of user event being notified by the Object class causeEvent, sdeCauseEvent, or sdsCauseEvent method, or the Class class causeClassEvent method.

The global constants (provided by the UserEvents category) for the range of user events that can be received are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any_User_Event</td>
<td>-1 (to receive to all user events)</td>
</tr>
<tr>
<td>User_Base_Event</td>
<td>16</td>
</tr>
<tr>
<td>User_Max_Event</td>
<td>Max_Integer (#7FFFFFFF, equates to 2147483647)</td>
</tr>
</tbody>
</table>

**theObject**

The theObject parameter specifies the object on which the event was caused.

**eventTag**

The eventTag parameter is the user-defined integer value that is used to identify a notification subscription that will be passed to the notification callback method.

**userInfo**

The userInfo parameter is a user-defined value of any primitive type (for example, String, Integer, or Character) that is received with each notification.

**User Notification Examples**

The following example shows a userNotification event that is performed when a specified event occurs; for example, supplies of a product are received.

```java
userNotification(eventType: Integer;  
    theObject: Object;  
    eventTag: Integer;  
    userInfo: Any) updating;

begin  
    if eventType = 17 then  
        displayProducts;
```
The following example, from the Erewhon Investments example schema supplied on the JADE release medium, shows the use of a `userNotify` method.

```plaintext
def userNotify(eventType: Integer; theObject: Object; eventTag: Integer; userInfo: Any)
  updating;
  begin
    app.mousePointer := Busy;
    zSynchronizeForm(eventType, theObject, eventTag, userInfo);
    epilog;
    app.mousePointer := Idle;
  end;
end;
```

The following example shows the use of a `userNotify` event that is executed when a notification registered for a user event is received.

```plaintext
def userNotify(eventType: Integer; theObject: Object; eventTag: Integer; userInfo: Any)
  updating;
  begin
    // The notification is identified using the eventType parameter, and an appropriate message is displayed to a text box.
    if eventType = 16 then
      textBox2.text := "User Class Notification Received";
      textBox7.text := userInfo.String;
    elseif eventType = 17 then
      textBox4.text := "User Notification Received";
      textBox8.text := userInfo.String;
  endif;
end;
```

**Ending a Notification**

Use the `endClassNotification`, `endClassesNotification`, `endClassNotificationForIF`, or `endClassesNotificationForIF` method to deregister a class notification and the `endNotification` or `endNotificationForIF` method to deregister an object notification registered by using the `beginNotification` or `beginNotificationForIF` method.

Alternatively, use the `endNotificationForSubscriber` method to terminate all previous notification requests for a specific subscriber.

**Note** The `endNotification`, `endClassNotification`, `endClassesNotification`, `endNotificationForIF`, `endClassNotificationForIF`, or `endClassesNotificationForIF` method `eventType` parameter must be the same as the `eventType` parameter specified in the `beginNotification`, `beginClassNotification`, `beginClassesNotification`, `beginClassNotificationForIF`, `beginClassesNotificationForIF`, or `beginNotificationForIF` method, respectively.
The parameters for methods that end notifications are listed in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventType</td>
<td>Specifies the type of event for which the notification subscription is to be terminated. This must be the same as the eventType parameters specified in the beginNotification, beginClassNotification, beginClassesNotification, beginClassNotificationForIF, beginClassesNotificationForIF, and beginNotificationForIF methods.</td>
</tr>
<tr>
<td>subscriber</td>
<td>For the endNotificationForSubscriber method, specifies the subscriber whose registered notifications are to be terminated.</td>
</tr>
<tr>
<td>theClass</td>
<td>For the endClassNotification, endClassesNotification, endClassNotificationForIF, and endClassesNotificationForIF methods, specifies the class for which the class notification of the specified type is to be terminated.</td>
</tr>
<tr>
<td>includeSubclasses</td>
<td>For the beginClassesNotification and beginClassesNotificationForIF methods, specifies whether subclasses are included or excluded in the notification registration.</td>
</tr>
<tr>
<td>theObj</td>
<td>For the endNotification and endNotificationForIF methods, specifies the object for which the notification subscription is to be terminated. Note that if this object is transient, only user end notifications can be invoked.</td>
</tr>
<tr>
<td>transients</td>
<td>For the endClassNotification, endClassesNotification, endClassNotificationForIF, and endClassesNotificationForIF methods, specifies if the user end notification is to be invoked for events that occur to transient instances (true), or persistent instances (false) of the class. (System notifications apply only to persistent objects.)</td>
</tr>
</tbody>
</table>

For more details, see the appropriate methods under "Object Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

For an example of the use of the endNotificationForSubscriber method, see "Receiving User Notifications" under "Receiving a Notification", earlier in this chapter.

The following example, from the Erewhon Investments example schema supplied on the JADE release medium, shows the use of the endNotification method when ending the notification on an item before removing it from the list.

```java
removeItem(pos : Integer) updating;
begin
  if showUpdates then
    if pos > 0 and pos <= self.listCount then
      endNotification(self.itemObject[pos], Object_Update_Event);
    endif;
  endif;
  inheritMethod(pos);
end;
```

The following example shows the use of the endClassNotification method when a form is unloaded.

```java
unload() updating;
begin
  endClassNotification(Sale, false, Object_Create_Event);
end;
```
The following example, from the Erewhon Investments example schema supplied on the JADE release medium, shows the use of the `endNotificationForSubscriber` method when clearing the contents of a list box and disabling notifications for the objects that were in the list.

```java
clear() updating;
begin
  // End all notifications
  endNotificationForSubscriber(self);
  // The list box is being cleared so clear our selected objects list
  zSelectedObjects.clear;
  if showUpdates and zCollectionOid <> null then
    // Turn notifications back on for the collection itself
    beginNotification(zCollectionOid.asOid.Collection, Any_System_Event,
                     Response_Continuous, NotifyCollectionUpdate);
  endif;
  inheritMethod;
end;
```

**Using Events**

JADE maintains lists of objects that have registered an interest in a specific event. When that event occurs, JADE invokes a predefined method for the appropriate object.

JADE uses notifications to send messages to an object that has previously requested a notification response by using the `beginNotification, beginClassNotification, beginClassesNotification, beginClassNotificationForIF, beginClassesNotificationForIF, or beginNotificationForIF` method.

For more details, see "Beginning a Notification", earlier in this chapter.

**Causing a Class Event**

Use the `causeClassEvent` method of the `Class` class to cause a class event. All objects subscribed to the specified event on instances of this class will be notified.

The parameters for the `causeClassEvent` method are listed in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventType</td>
<td>An integer value selected by the user, in the range 16 through <code>User_Max_Event</code>.</td>
</tr>
<tr>
<td>immediate</td>
<td>Value to indicate when the event is actioned. If this value is <code>false</code>, the notification occurs at the end of the transaction. If the client is not within a begin/commit transaction cycle, the notification waits for the next commit on that client. (If the value is <code>true</code>, the notification occurs immediately.)</td>
</tr>
<tr>
<td>userInfo</td>
<td>Value of type <code>Any</code> that is passed to the <code>userNotify</code> or <code>userNotification</code> event handler when the notification is received. Notifications containing binary and string (Binary, String, StringUtf8) data of up to 48K bytes can be sent across the network. For applications running within the server node, the limit for notifications containing binary or string data is 2G bytes. Note, however, that this applies only to single user and server applications. In multiuser applications, persistent notifications are sent via the database server, even if the receiving process is on the same node as the sender. In notification cause events, exception 1267 (Notification info object too big) is raised if the binary or string <code>userInfo</code> data exceeds the applicable value.</td>
</tr>
</tbody>
</table>
The following example shows the use of the `causeClassEvent` method, where a notification is sent immediately.

```pascal
userButton_click(btn: Button input) updating;
begin
  Customer.causeClassEvent(Refresh_Customer_Views, true, name);
end;
```

For more details about the `causeClassEvent` method, see "causeClassEvent" under "Class Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

**Causing an Object Event**

Use the **Object** class `causeEvent` method to trigger a user event, the `sdeCauseEvent` method for inter-system event notification in a Synchronized Database Environment (SDE), or the `sdsCauseEvent` method for inter-system event notification in a Synchronized Database System (SDS).

The `sdeCauseEvent` method combines the actions of the `causeEvent` and `sdsCauseEvent` methods, in that subscribers are notified of user events on the local system as well as on SDE secondary or primary systems, where applicable. In contrast, the `causeEvent` method would notify subscribers of a user event only on the primary database system and the `sdsCauseEvent` method only on the secondary database systems.

A corresponding notification is received by any objects that have registered an interest in the notification by using a begin notification method for that object or its class. (For more details, see "causeEvent", "sdeCauseEvent", or "sdsCauseEvent" under "Object Class", in Chapter 1 of the JADE Encyclopaedia of Classes.)

The parameters for the `causeEvent` method are listed in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>eventType</code></td>
<td>Integer in the range <code>User_Base_Event</code> through <code>User_Max_Event</code> that represents the event being caused; that is, a number in the range 16 through <code>User_Max_Event</code>.</td>
</tr>
<tr>
<td><code>immediate</code></td>
<td>Boolean value specifying the timing of the event; <code>false</code> indicates that notifications occur at the end of transaction and <code>true</code> indicates that the notification is sent immediately. If the client is not within a begin/commit transaction cycle and this parameter is set to <code>false</code>, the notification waits for the next commit on that client.</td>
</tr>
<tr>
<td><code>userInfo</code></td>
<td>A value of any primitive type value (for example, a <code>String</code> or an <code>Integer</code>) or persistent object reference that is passed to the <code>userNotification</code> or <code>userNotify</code> event handlers when the event is notified. (Notifications containing binary and string (<code>Binary</code>, <code>String</code>, <code>StringUtf8</code>) data of up to 48K bytes can be sent across the network. For applications running within the server node, the limit for notifications containing binary or string data is 2G bytes. Note, however, that this applies only to single user and server applications. In multiuser applications, persistent notifications are sent via the database server, even if the receiving process is on the same node as the sender. In notification cause events, exception 1267 (<code>Notification info object too big</code>) is raised if the binary or string <code>userInfo</code> data exceeds the applicable value.) Although you should not use a transient object reference across nodes, you can use a shared transient object reference between applications on the same node.</td>
</tr>
</tbody>
</table>

The following example shows the use of the `causeEvent` method to set off a user notification for the `b1` instance of the `B` class. The notification is given an event type of `User_Base_Event` to identify it, and the `userInfo` parameter is passed the contents of a text box. The `userInfo` parameter is passed into the `userNotify` event when the notification is received.

```pascal
btnClassUserEvnt_click(btn: Button input) updating;
begin
```

```pascal
```

```pascal
DevRef - 2018.0.01
b1.causeEvent(User_Base_Event, true, textBox5.text);
end;

The following example shows the use of the `causeEvent` method to set off a user notification for the `c2` instance of the `C` class. The notification is given an event type of 17 to identify it, and the `userInfo` parameter is passed the contents of a text box. The `userInfo` parameter is passed into the `userNotify` event when the notification is received.

```
btnUserEvnt_click(btn: Button input) updating;
begin
  c2.causeEvent(17, true, textBox6.text);
end;
```

The following example shows the use of the `causeEvent` method to reduce the price of shares for a company by 10 percent.

```
discountPrice() updating;
begin
  self.priorPrice := currentPrice;
  // don't reduce the price below 20 cents
  if currentPrice > 0.20 then
    self.currentPrice := currentPrice *.90;
  endif;
  causeEvent(Price_Change, false, 0);
end;
```

The following example shows the use of the `causeEvent` method to notify other applications in the current node that this application is closing down.

```
finalize() updating;
begin
  endNotificationForSubscriber(self);
  myInterProcessCommunication.causeEvent(Application_Closing_Down,
                                          true, 0);
  // myInterProcessCommunication is a shared transient object used
  // to communicate with all applications in the node
end;
```

The behavior of the `sdsCauseEvent` method is database role-dependent. The three database role categories are listed in the following table.

<table>
<thead>
<tr>
<th>Role</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>When invoked within an SDS primary system, the <code>sdsCauseEvent</code> method audits the event for subsequent replay by SDS secondary databases. The event is not notified on the primary. The value of the <code>immediate</code> parameter must be <code>false</code>.</td>
</tr>
<tr>
<td>Secondary</td>
<td>When invoked within an SDS secondary system connected to a primary database server, the <code>sdsCauseEvent</code> method triggers a corresponding event on the same receiver object in the primary system. The user event is not notified on the secondary system. Events caused on a secondary are assumed to be immediate, so the <code>immediate</code> parameter is therefore ignored.</td>
</tr>
<tr>
<td>None</td>
<td>When invoked within a non-SDS-capable system (that is, the role is undefined), the method behavior is the same as the <code>Object::causeEvent</code> method.</td>
</tr>
</tbody>
</table>
The parameters for the `sdsCauseEvent` method are listed in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventType</td>
<td>Integer in the range <code>User_Base_Event</code> through <code>User_Max_Event</code> that represents the event being caused; that is, a number in the range 16 through <code>User_Max_Event</code>.</td>
</tr>
<tr>
<td>immediate</td>
<td>You must set this parameter to false when the method is invoked from a primary system in a Synchronized Database Environment (SDE).</td>
</tr>
<tr>
<td>userInfo</td>
<td>A value of any primitive type value (for example, a <code>String</code> or an <code>Integer</code>) or persistent object reference that is passed to the <code>userNotification</code> or <code>userNotify</code> event handlers when the event is notified. (Notifications containing string and binary data of up to 48K bytes can be sent across the network.)</td>
</tr>
</tbody>
</table>

The parameters for the `sdeCauseEvent` method are listed in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eventType</td>
<td>Integer in the range <code>User_Base_Event</code> through <code>User_Max_Event</code> that represents the event being caused.</td>
</tr>
<tr>
<td>immediate</td>
<td>Boolean value specifying the timing of the event; false indicates that notifications occur at the end of transaction and true indicates that the notification is sent immediately. If the client is not within a begin/commit transaction cycle and this parameter is set to false, the notification waits for the next commit on that client. On a primary database, subscribers are notified only on the secondaries if the target object is persistent, the value of the immediate parameter is false, and the process is currently in transaction state. On a secondary database system, subscribers are notified on the primary database system only if the target object is persistent. The value of the immediate parameter is immaterial. However, subscribers are always notified immediately on the primary database system, even when the value of the immediate parameter is false, to defer notification on the secondary system.</td>
</tr>
<tr>
<td>userInfo</td>
<td>A value of any primitive type value (for example, a <code>String</code> or an <code>Integer</code>) or persistent object reference that is passed to the <code>userNotification</code> or <code>userNotify</code> event handlers when the event is notified. (Notifications containing string and binary data of up to 48K bytes can be sent across the network.)</td>
</tr>
</tbody>
</table>

For more details about inter-system communications in an SDE, see "Inter-System Event Notifications", in Chapter 10.
Chapter 3  
Exceptions

This chapter covers the following topics.

- Overview
  - Error Handling Policy
  - System Exceptions
  - User Exceptions
- Handling Exceptions
  - Automatic Updates and Exception Handling
- Dealing with System Exceptions
  - Default Exception Handler
- Creating and Arming an Exception Handler
  - Creating an Exception Handler
  - Arming an Exception Handler
  - Disarming a Local Exception Handler
  - Disarming a Global Exception Handler
  - Viewing the Exception Handler Stack
  - Using Exception Handlers in Non-JADE Applications
- Using the raise Method
- Using the raise exception Instruction
- Interpreting Exceptions
- Monitoring Forms to Identify an Exception Dialog

Overview

In a complex system, you would generally write different parts of a JADE application independently. The JADE application can be based around an existing framework or it can make use of existing services in a library or class hierarchy.

Errors are represented and handled in JADE by using exception objects (instances of the Exception class or its subclasses).

JADE assumes that errors or exceptions in a stable production system will occur infrequently (if at all), but they cannot be ignored completely. Error handling code is therefore separated from the mainstream application code, providing cleaner application code that is not cluttered with error handling code.

JADE exception handling applies to both JADE methods and external methods, and provides a mechanism that enables you to construct JADE applications that are tolerant of exceptions that occur because of abnormal conditions.
At run time, the subsystem providing a service can detect an error but will generally not know what to do about the error. Alternatively, the subsystem using the service may be able to deal with the error but may not be able to detect it.

The Exception class and its subclasses are used to deal with system exceptions and raising user exceptions in both JADE methods and external methods. Other error conditions (for example, errors in user input) fall into the category of normal application code that implements a business rule.

Notes In JADE thin client mode, exception dialogs are always displayed on the presentation client.

For details about finding the position in a method source if you want to locate the position at which an exception occurred in a deployed application from which source code has been removed, see "Finding a Method Source Position", in Chapter 4 of the JADE Development Environment User’s Guide.

Call the Object class getClassForObject method if you want to return a reference to the class of a specified object identifier (oid) even if this object is no longer valid; for example, in exception handlers that may need to deal with object references that are no longer valid.

For a discussion about exception handling, see the Erewhon.pdf file that is included in the erewhon subdirectory of the examples directory on the JADE release medium.

Exception states on client nodes and server nodes are independent. If an exception occurs while executing on the client node, exception state on the server node is not set. Similarly, if an exception occurs while executing a server execution method, exception state on the client node is not set.

For details about using generic exit values that apply to JADE programs and utilities when running JADE, see Appendix C, in Appendix A of the JADE Installation and Configuration Guide. These exit values enable you to develop tools that can take appropriate actions based on the exit values of supplied programs. In addition, a range of exit values is set aside for warnings, enabling a program to return information without it being regarded as a fatal problem.

Error Handling Policy

The detection of an abnormal condition provides the user with the following options.

- **Retry;** that is, attempt to change the conditions that led to the exception and retry the operation or fix the problem, and then continue execution from the point at which the error was detected. This approach is often applicable to database lock contention.

- **Termination;** that is, clean up or return the environment to a stable state (abort transaction), report the failure with an error message, and then terminate execution. This is a more conservative, and generally safer, approach.

Regardless of which approach you take to abnormal error conditions, an operation can succeed or fail only in its entirety.

System Exceptions

System exceptions are error conditions that are raised automatically by the system; for example, a lock exception or an integrity violation.

Note A 1092 (Object not available) exception may be raised during an exception handling operation if you are trying to access the object that caused the exception and that object is not yet read into cache.

If this exception is raised, fix your application code so that objects that cause an exception during their access (usually lock operations) are not referenced.
There is more sensitivity in a "dirty" read of uncommitted changes to a persistent object when both processes run in the same node, because dirty reads are detected inside the common node while dirty reads across nodes are detected on the server node. Processes that share the same node are therefore likely to get more dirty read exceptions. Use lock exception handling in your code to deal with dirty read situations that may occur.

For details about sharing uncommitted persistent objects, see "Sharing Uncommitted Persistent Objects", in Chapter 1. For details about dirty reads, see "JADE Object Manager Distributed Processing", in Chapter 1 of the JADE Object Manager Guide.

### User Exceptions

You can define exceptions other than those automatically captured by the system. This enables you to add new properties and methods specific to your own exception protocol or to override system methods.

**Tip**  As JADE itself uses exception codes with lower numbers (that is, those less than 63,999), you should define error codes for your user-defined exceptions in the range 64000 through Max_Integer (#7FFFFFFF, which equates to 2147483647).

### Handling Exceptions

In the absence of an exception handler, the default Unhandled Exception dialog is displayed when an exception is raised and the exception (including the method call stack history and the exception stack history) is written to the log file of your current application; for example, **MyApp.log**. This dialog provides details of the current exception, and buttons enabling you to:

- Abort
- Debug
- Ignore (continue)

Pressing the Esc key is equivalent to clicking the **Abort** button.

The unhandled exception dialog and the logged unhandled exception details include the application and schema names. In addition, the logged exception details contain the computer name.

You can define an exception handler method to override the default exception handler.

When an invalid code is returned from an exception handler (for example, when resuming a non-resumable exception or continuing a non-continuable exception), a **1238 (Exception Handler invalid return code)** exception is raised.

When the maximum number of nested exceptions is reached (this limit is currently 20), an exception is raised. For this type of exception, no user exception handler is called but the **defaultHandler** method of the **Exception** class is invoked. This **defaultHandler** method is the "handler of the last resort", which is automatically invoked on the exception instance if no other exception handler consumes the exception. The exception, including the method call stack history and the exception stack history, is written to the log file of the current application.

Your exception handler should:

- Test to see if an exception is continuable before attempting to return **Ex_Continue**.
- Include checks to see if it is a nested exception situation.
- Specifically check for the 1238 exception.
In client-side GUI applications, the `defaultHandler` method calls the `Exception` class `showDialog` method on `self`. If the `showDialog` method returns `true`, the `defaultHandler` method ignores the exception by returning the result of `Ex_Continue`.

**Note** It is not valid to ignore non-continuable exceptions. However, if the `showDialog` method returns `false` (indicating that the user clicked the `Abort` button), the `defaultHandler` method first aborts any current persistent or transient transaction and then terminates the current action by returning an `Ex_Abort_Action` result.

When the `defaultHandler` method is invoked for an exception raised in non-GUI-capable methods (including server methods, server application methods, and any non-GUI application methods), the `defaultHandler` method does not call the `showDialog` method but instead aborts any current persistent or transient transaction, logs the exception to the exception log file of the current application, and when invoked from a server method it returns `Ex_Pass_Back`; otherwise, it returns `Ex_Abort_Action`.

**Notes** You can reimplement the `defaultHandler` method in your subclasses of the `Exception` class if you want different default behavior.

Your exception handlers that do not handle a specific exception should return `Ex_Pass_Back` rather than directly calling the `defaultHandler` method.

The `Exception::showDialog` method displays the default exception dialog. The `showDialog` method returns `true` if the user clicks the `Ignore` button in the default exception handler and the exception is continuable, otherwise this method returns `false`. The `Ignore` button is disabled if the exception is not continuable or if production mode is enabled for the database.

The `showDialog` method can be called from a user-defined exception handler and can be called by the `defaultHandler` method.

In situations where an `Exception handler invalid return code` exception is raised, the offending exception handler is not displayed because it has already finished its execution but the `resumable` and `continuable` properties of each exception on the exception stack is logged in the exception stack history section of the log file of your current application.

Each process can have up to 128 global exception handlers armed at any one time. However, there is no JADE-imposed restriction on the number of non-global exception handlers that can be armed at any time by any one process.

JADE does not allow resuming from global exception handlers, because there is no guarantee that the method that armed the exception handler is still in the call stack. Using `Ex_Resume_Next` in a global exception handler is equivalent to returning `Ex_Abort_Action`.

As the deletion of objects is atomic, transactions are aborted when exceptions are raised during automatic maintenance actions. If the attempt to commit a transaction after an exception occurs during a delete action (for example, user code in a `delete`, or destructor, method), the transaction is aborted if any user-defined exception handler does not do so. No objects are deleted after the exception.

When an exception occurs in a server method and no exception handler was armed on the server node for the exception, JADE performs the following actions.

1. The default exception handler is invoked for the exception. If this is a JADE exception handler, the application stack is written to the application log file but the corresponding dialog is not displayed.
2. An exception `1242 (A method executing in another node was aborted)` is raised on the client node that invoked the server method execution. The `extendedErrorText` property of this exception contains the text that corresponds to the original server method exception.

To handle exceptions in server methods correctly, exception handlers can be armed in server methods themselves. These exception handlers can be local or global.
Chapter 3 Exceptions

Note Global exception handlers armed in methods executing on the client node have effect on client exceptions only and global exception handlers armed in methods executing on the server node have effect only on the server node.

When an exception handler is invoked from a package method, the package context is switched to the context that was in effect when the exception handler was armed. When the exception handler finishes, the package context is adjusted back, if required.

See also "Default Exception Handler", later in this chapter. For a list of the global constants that you can use in your exception handlers, if required, see the JadeErrorCodesDatabase, JadeErrorCodesSDS, and JadeErrorCodesWebService global constants categories in Appendix A of your JADE Encyclopaedia of Classes.

Automatic Updates and Exception Handling

If an exception occurs during an automatic update (for example, inverse maintenance, automatic key maintenance, or parent-child deletion), the transaction is aborted when returning from the first exception handler. The transaction is aborted because allowing the transaction to be committed could result in updating one side of a relationship (the property or collection whose update initiated the automatic update) but not updating the other side (because that operation encountered an exception). This could result in a loss of logical integrity caused by corrupt inverses or dictionary keys.

Note While the exception handler is executing, you are still in transaction state and objects created or updated in the transaction are still visible. Any attempt to commit the current transaction in your exception handler raises an exception 1252 (Transaction must be aborted). If you abort the current transaction in the exception handler, which is often a good idea, you can then begin and commit new transactions.

The aborting of a transaction when an automatic update cannot be completed may affect the way you choose to code a task. For example, if you create a large number of objects in a single transaction, an automatic update exception (for example, 1310 (Key already used in this dictionary) can be raised if the objects are being added to a dictionary as part of the automatic inverse maintenance.

Aborting the transaction results in all previously created objects being lost. In such situations, you may choose to check for the existence of a duplicate key by calling the Dictionary class includesKey method rather than using an exception handler to deal with any duplicates.

Dealing with System Exceptions

JADE raises a number of exceptions automatically. These are known as system exceptions. In JADE, exceptions are implemented as objects.

The following table shows the Exception class of the RootSchema and its subclasses.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception</td>
<td>Superclass of all exceptions</td>
</tr>
<tr>
<td>FatalError</td>
<td>Serious internal fault</td>
</tr>
<tr>
<td>NormalException</td>
<td>Superclass of non-fatal exceptions</td>
</tr>
<tr>
<td>ConnectionException</td>
<td>Defines behavior for exceptions that occur as a result of connecting to external systems</td>
</tr>
<tr>
<td>FileException</td>
<td>Class of exception raised as a result of file handling</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>JadeMessagingException</td>
<td>Defines behavior for exceptions that occur when using the JADE messaging framework</td>
</tr>
<tr>
<td>JadeSOAPException</td>
<td>Defines behavior for exceptions that occur as a result of Web services processing</td>
</tr>
<tr>
<td>JadeXMLException</td>
<td>Defines behavior for exceptions that occur as a result of XML processing</td>
</tr>
<tr>
<td>ODBCException</td>
<td>Defines behavior for exceptions that occur as a result of connecting to external databases using ODBC</td>
</tr>
<tr>
<td>SystemException</td>
<td>Superclass of all exceptions relating to errors detected by the JADE Object Manager</td>
</tr>
<tr>
<td>DeadlockException</td>
<td>Exception raised as a result of a deadlock</td>
</tr>
<tr>
<td>IntegrityViolation</td>
<td>Exception raised as a result of violating referential integrity rules</td>
</tr>
<tr>
<td>LockException</td>
<td>Exception raised as a result of locking conflicts in a multiuser environment</td>
</tr>
<tr>
<td>NotificationException</td>
<td>Exception raised as a result of a notification event when the subscriber cannot be found</td>
</tr>
<tr>
<td>UserInterfaceException</td>
<td>Class of exception related to handling windows</td>
</tr>
<tr>
<td>ActiveXInvokeException</td>
<td>Defines behavior for exceptions that occur as a result of accessing an ActiveX property or invoking an ActiveX method</td>
</tr>
<tr>
<td>JadeDotNetInvokeException</td>
<td>Defines behavior for exceptions that occur as a result of accessing a .NET property or invoking a .NET method</td>
</tr>
</tbody>
</table>

You can add further Exception subclasses in your subschemas to create exceptions specific to an application.
Default Exception Handler

By default, the `defaultHandler` method of the `Exception` class is invoked when an exception is raised and there is no appropriate exception handler armed. This method displays a dialog showing details of the current exception, as shown in the following example.

The `Abort` button aborts the action, effectively terminating the code that was executing at the time of the exception.

The `Ignore` button ignores the exception and continues execution from the next expression after the expression that caused the exception. This may not always be desirable, as the failure of one instruction may lead to further exceptions or incorrect results. The `Ignore` button is disabled if the exception cannot be continued or if production mode is enabled for the JADE database. As most system exceptions are unable to be continued, the `Ignore` button is usually disabled for system exceptions.

See also "Handling Exceptions", earlier in this chapter.
Debugging the Method Call Stack

The Debug button displays a Call Stack Browser. The Call Stack Browser enables you to browse methods in the call stack, and inspect parameters and local variables.

**Note** The Call Stack Browser is also displayed when you call the Exception class debug method in your own exception handlers.

The exception, including the method call stack history and the exception stack history, is also written to the log file of the current application (for example, MyApp.log). The exception stack history section contains at least the error code of each of the nested exceptions.

The following image shows an example of the Call Stack Browser, with the EconAgent class ad1 property File class Mode_Input constant selected for inspection.

> Use the Inspect Item command from the Inspect menu to display the value of a primitive item or to inspect an item containing an object reference.

**To copy the current call stack to the clipboard as a string**

1. Right-click on the call stack results.
2. Select the Copy Call Stack to Clipboard command from the popup menu that is then displayed.
To inspect the value of the item on which the caret is positioned

- Perform one of the following actions.
  - Select the Inspect Item command from the Inspect menu
  - Press Ctrl+I

If the item being inspected contains a primitive type, an invalid object reference, or a null object reference, a dialog is then displayed. (An example of this dialog is shown in the previous image.)

If the item contains an object reference, an Inspector form for the object is then displayed. The selected object, and any properties defined for that object, is displayed in the left of the Inspector form.

To view the values in the Inspector form for the displayed properties

- Click on the required property.

The values for the property are then displayed in the pane at the right of the form.

For more details, see Chapter 7 of the JADE Development Environment User's Guide, "Using the JADE Debugger".

To exit from the Call Stack Browser

- Select the Exit command from the File menu.

The Call Stack Browser is then closed, and focus returns to the Unhandled Exception dialog.

Creating and Arming an Exception Handler

When creating and arming an exception handler, the term action is used to describe the processing steps that take place between the point that a user causes some event that is processed by the system to the point at which the system is again in a quiescent state waiting for another event.

Each called method can have a maximum of 128 armed local exception handlers at the same time. Exceeding this limit raises an exception. This limit applies to each called method rather than being an overall maximum.

Creating an Exception Handler

In most cases, you will want to take some action other than the default when a system exception occurs. To do this, you must define and arm an exception handler.

An exception handler is a method that can have the following signature.

```
method-name(e: Exception-class; [other-optional-parameters]): Integer;
```

In this signature, the `e` parameter refers to the exception object created and passed when the exception was raised.

You can optionally specify other parameters when you are creating or arming a method handler to suit your requirements, but you must specify the parameter that refers to the exception object as the first parameter.

The exception handler would normally use information saved in the `Exception` class object to decide the action to be taken. Within your method, you can use the `isKindOf` method to determine the class of the exception object.

An exception handler can also have the following signature.

```
method-name(exObj: any-exception-class; [other-parameters]): Integer;
```
In this signature, the `exObj` parameter indicates that this exception handler handles only exceptions that are instances of the specified exception class or its subclasses. You could specify `o` and `i` optional parameters, for example, to indicate the current object and an integer variable, respectively.

**Note** In these exception handler method signatures, the `e` or `exObj` parameter must be the first parameter that you specify in the exception handler method signature.

The contents of the parameter expressions passed to the exception handler method are evaluated as follows.

- **When the exception is armed for global exception handlers**, as shown in the following example.

```plaintext
exampleMethod;
vars
    obj : Object;
    thing : String;
begin
    obj := global;
    thing := "very first time";
    on Exception do ehWithParameters(exception, thing, obj) global;
    thing := "first time";
    /* if an exception is caused by the following line of code, the
     * thing parameter contains the value "very first time" and the
     * obj parameter refers to global */
    m2;
    obj := app;
    thing := "second time";
    /* if an exception is caused by the following line of code, the
     * thing parameter contains the value "very first time" and the
     * obj parameter refers to global */
    m2;
end;
```

- **For local exception handlers**, at the moment that the exception occurs, as shown in the following example.

```plaintext
exampleMethod;
vars
    obj : Object;
    thing : String;
begin
    obj := global;
    thing := "very first time";
    on Exception do ehWithParameters(exception, thing, obj);
    thing := "first time";
    /* if an exception is caused by the following line of code, the
     * thing parameter contains the value "first time" and the obj
     * parameter refers to global */
    m2;
    obj := app;
    thing := "second time";
    /* if an exception is caused by the following line of code, the
     * thing parameter contains the value "second time" and the obj
     * parameter refers to app */
    m2;
end;
```
The return value indicates the action that the system then takes, as shown in the following table.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Constant</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ex_Continue</td>
<td>Continues execution from the next expression after the expression that caused the exception.</td>
</tr>
</tbody>
</table>

**Caution**  Use this return mode only in circumstances when you are certain that continuing the code execution will still be correct after ignoring the exception.

For lock exceptions, use this return mode only if the lock has been successfully retried. If the exception occurred while updating, ensure that the transaction has not been aborted by the exception handler.

| 1            | Ex_Abort_Action  | Causes the current action to be aborted. The execution stack is stripped back and in most cases, the application reverts to an idle state in which it is waiting for user input or some other Windows event (see the note following this table). |

| 2            | Ex_Resume_Next   | Passes control back to the method that armed the exception handler. Resumes from the next statement after the evaluation of the method call or expression in which the exception occurred. If there were no messages on the execution stack when the handler was armed, the effect of the Ex_Resume_Next call is identical to that of the Ex_Abort_Action call. |

**Note**  If an exception occurs during evaluation of a return expression, when the exception handler returns, the executing method is terminated (in accordance with executing a return instruction) and the default value of the return type is returned.

| -1           | Ex_Pass_Back     | Passes control back to the prior local exception handler for this type of exception or if a local handler is not found, a global exception handler for this type of exception. If neither a local nor a global handler is found, the default exception handler is invoked on a client. On the server, it passes the exception back to the client node. |

Trying to continue a non-continuable exception causes a further exception 1238 (Exception handler invalid return code) to be raised. If this exception is caught by an exception handler that then tries to continue the exception, an exception 1239 (Nested exceptions limit exceeded) is eventually raised, due to repeated 1238 exceptions. Your exception handlers should therefore test to see if an exception is continuing or not before attempting to return Ex_Continue. Your exception handler should also include checks to see if it is in a nested exception situation. It may also be beneficial to specifically check for the 1238 exception.

If an exception occurs and an exception handler method catches the exception and does an Ex_Resume_Next, the stack is cut back to the method that armed the exception handler and execution continues normally from this point.

However, if a second exception that is also handled by an exception handler that does an Ex_Resume_Next occurs while the stack is being cut back, after the stack has been cut back to the method that armed this (second) exception handler, the cutting back of the stack continues to the method that armed the first exception handler.
Notes Any database transaction that is in progress is not aborted. You must explicitly code an abortTransaction instruction within the exception handler if the database transaction in progress is also to be aborted.

If an exception occurs while executing a server method and no user exception handlers are armed, the default handler is invoked on the server node. This aborts any current transaction on the server and passes a 1205 exception back to the client node.

You can use the methods listed in the following table in your exception handlers.

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exception</td>
<td>logExceptionHistory</td>
<td>Enables you to log the exception stack history in an exception handler to the specified file</td>
</tr>
<tr>
<td>Exception</td>
<td>logProcessHistory</td>
<td>Enables you to log the call stack history in an exception handler to the specified file</td>
</tr>
<tr>
<td>Exception</td>
<td>logSelf</td>
<td>Appends a description of the exception object to the specified file</td>
</tr>
<tr>
<td>JadeMethod</td>
<td>getSourceLine</td>
<td>Returns the line of JADE code that contains the specified source position, to provide diagnostics during JADE development in your user exception handlers</td>
</tr>
<tr>
<td>MethodCallDesc</td>
<td>logSelf</td>
<td>Enables you to log diagnostic information from an exception handler</td>
</tr>
</tbody>
</table>

For details, see Chapter 1 of the JADE Encyclopaedia of Classes.

The following is an example of a log file containing output from the exception logging methods.

```
theLibrary.log

Computer= WILBUR1A, Application= TheLibrary, Schema= TheLibrary
SystemException: 1048 (not continuable, resumable)
 * Update outside transaction
   Error object: Address/3798.1
   Caused By:
     Receiver: Address/3798.1
     Method: Address::loadAddress(117) -- adrLinel := adl.trimBlanks ;
     // copy the values passed from the form

   Call Stack History:
     <<Address/3798.1>> Address::loadAddress(117) -- adrLinel := adl.trimBlanks ;
     // copy the values passed from the form
     <<JadeScript/107.1 (t)>> editTester(396) -- adr.loadAddress

Exception:- Type: SystemException; Error Code: 1048 on 2018/01/23 12:57:49
*** User ABORTED current action and database transaction ***
```
Examples of Defining an Exception Handler

The following example shows the definition of an exception handler method that has only the mandatory parameter (ex, in this example) that refers to the NormalException object.

```pascal
exceptionHandler(except: NormalException):Integer;
vars
  number : Integer;
begin
  // This is the exception handling method that is specified when arming
  // the handler. It will be called when the appropriate exceptions are
  // raised.
  // If the error code of the exception is 1035, the exception is
  // identified as being the String Too Long exception and is handled
  // appropriately.
  if except.errorCode = 1035 then
    number := app.msgBox("String too long. Resume execution after method?", "String too long", 52);
    if number = MsgBox_Return_Yes then
      // The Ex_Resume_Next return value will pass control back to the
      // method that armed the exception handler (in this case,
      // method1) and resume execution after the invocation of the
      // method that raised the exception.
      return Ex_Resume_Next;
    else
      // The Ex_Abort_Action return value will cause all currently
      // executing methods to be aborted. In this case, the
      // application will revert to execution after the invocation of
      // the method that raised an idle state, and await further user
      // input.
      status.caption := "Aborting all currently executing methods";
      return Ex_Abort_Action;
    endif;
  // If the error code of the exception is 64000, the exception is
  // identified as being the user-defined exception that was assigned this
  // code, and is handled appropriately.
  elseif except.errorCode = 64000 then
    number := app.msgBox("User-defined exception. Continue method execution?", "User-defined exception", 52);
    if number = MsgBox_Return_Yes then
      // The Ex_Continue return value will pass control back to the
      // method that raised the exception handler (in this case,
      // method4) and resume execution after the raising of the
      // exception.
      return Ex_Continue;
    else
      // The Ex_Resume_Next return value will pass control back to the
      // method that armed the exception handler (in this case,
      // method1) and resume execution after the invocation of the
      // method that raised the exception.
      status.caption := "Resuming execution after exception throwing method invocation";
      return Ex_Resume_Next;
    endif;
  endif;
end;
```
Chapter 3  Exceptions

For examples of the methods used to arm the exception handler defined in the above method, see "Examples of Arming an Exception Handler", later in this chapter. For an example of the method that creates an object of the UserException class, defines the properties for this object, and then raises the exception, see "Using the raise exception instruction", later in this chapter.

The following example shows the definition of an exception handler method that has only the mandatory parameter (exObj, in this example) that refers to the exception object.

```pascal
handleException(exObj: Exception): Integer;
  constants
    StringTooLong = 1035;
  begin
    if exObj.errorCode = StringTooLong then
      app.msgBox("The picture is too large: ", "Error", MsgBox_OK Only);
      return Ex_Abort_Action;
    else
      return Ex_Pass_Back;  // pass exception to prior handler
  end;
```

### Arming an Exception Handler

When you have defined an exception handler, you must then arm it. The following syntax is used to arm an exception handler.

```pascal
on exception-class do method-call-expression [global];
```

The exception-class identifier is the Exception class or one of its subclasses. Use the optional global system variable to arm a global exception handler. As a global exception handler is not associated with a specific method, it is always armed until it is explicitly disarmed. A local exception handler is automatically disarmed when the method that armed it returns.

**Note** Each process can have up to 128 global exception handlers armed at any one time. There is no JADE-imposed restriction on the number of non-global exception handlers that can be armed at any time by any one process.

When you arm an exception handler, note the following points.

- The method-call-expression identifier can be in the following form.

  ```pascal
  object-reference-expression.method-identifier
  ```

  In this case, method-identifier is invoked for the specified object if an exception occurs.

- The method-call-expression identifier can be in the form method-identifier, in which case method-identifier is invoked for the object upon which the method arming the exception handler is invoked.

- The method used to arm an exception handler can have parameters in addition to the exception parameter that refers to the exception object created and passed when the exception was raised. For example, you could specify an optional parameter that refers to the current object and an optional parameter that refers to an integer variable; for example:

  ```pascal
  on Exception do exceptionHandler(exception, currentObj, n);
  ```

  The exception object must be the first parameter that you specify when arming an exception handler.

- As the receiver of an exception must be an object, you cannot arm the exception handler when the receiver is a primitive type.
The contents of the parameter expressions passed to the exception handler method are evaluated as follows.

- At the moment the exception handler is armed, for global exception handlers
- At the moment that the exception occurs, for local exception handlers

### Examples of Arming an Exception Handler

The following two examples show the methods required to arm the `exceptionHandler` method in the first of the examples under "Examples of Defining an Exception Handler", earlier in this chapter. (For an example of the method that creates an object of the `UserException` class, defines the properties for this object, and then raises the exception, see "Using the `raise exception` instruction", later in this chapter.)

```pascal
method1();
begin // Arms the exception handler so that the exceptionHandler method will
  // be called when an exception of the NormalException class is
  // encountered and will be passed the exception object as a parameter.
  on NormalException do exceptionHandler(exception);
  // causes an exception
  method2;
  status.caption := "Resuming execution after exception throwing method
  invocation";
end;

method3();
begin // Arms the exception handler so that the exceptionHandler method will
  // be called when an exception of the UserException class is raised and
  // will be passed the exception object as a parameter.
  on UserException do exceptionHandler(exception);
  method4;
end;
```

The following is an example of a method on a `Document` class that writes the document to a file and arms a local exception handler to catch file exceptions.

```pascal
writeToFile();
vars
  fileSaveDialog : CMDFileSave;
  file : File;
begin
  on FileException do fileExceptionHandler(exception);
  create fileSaveDialog transient;
  if fileSaveDialog.open = 0 then
    create file transient;
    file.mode := File.Mode_Output;
    file.fileName := fileSaveDialog.fileName;
    saveToFile(file);
  file.close;
  delete file;
end;
```

The following lines demonstrate the use of an exception handler to catch and process file exceptions when saving a document to a file.

```pascal
writeToFile();
vars
  fileSaveDialog : CMDFileSave;
  file : File;
begin
  on FileException do fileExceptionHandler(exception);
  create fileSaveDialog transient;
  if fileSaveDialog.open = 0 then
    create file transient;
    file.mode := File.Mode_Output;
    file.fileName := fileSaveDialog.fileName;
    saveToFile(file);
  file.close;
  delete file;
end;
```
Disarming a Local Exception Handler

A local exception handler normally remains armed for the duration of the method that armed it. The following syntax disarms a local exception handler of a specified exception class.

```
on exception-class do null;
```

This disarms the last (or most recently armed) exception handler for exceptions of the specified exception class.

Disarming a Global Exception Handler

A global exception handler remains armed until it is explicitly disarmed or the process that armed it terminates. The following syntax disarms a global exception handler.

```
on exception-class do null global;
```

Viewing the Exception Handler Stack

You can execute the `getExceptionHandlerStack` method of the `Process` class to populate a transient array with transient `ExceptionHandlerDesc` objects that represent the exception handlers armed by the current process in the current node.

In the array, locally armed exception handlers precede globally armed exception handlers. Within this grouping, the most recently armed exception handlers occur first.

The following example shows the use of the `getExceptionHandlerStack` method to display the exception handler stack.

```
show Armed Exception Handlers()
vars
   oa: ObjectArray;
   o: Object;
beginn
   create o transient;
   process.getExceptionHandlerStack(oa);
   foreach o in oa do
      write o.display;
   endforeach;
end;
```

The `getExceptionHandlerStack` method can be called only on the current process.

Using Exception Handlers in Non-JADE Applications

You can define and arm exception handlers in non-JADE applications (that is, in applications written in Smalltalk, Visual Basic, or C++ that interface to the JADE Object Manager Application Programming Interface).

Non-JADE applications should arm an exception handler by calling the JADE Object Manager API `jomArmExceptionHandler` entry point. For details about the parameters to this call, see "Arming an Exception Handler", in Chapter 3 of the JADE Object Manager Guide.
Note: As the default exception handler is not invoked if you intercept an exception with your own handler, the Unhandled Exception dialog is not displayed, regardless of the value returned for your exception handler.

If you want to invoke the default handler and therefore display the Unhandled Exception dialog and write the exception to the application log file, you must send the message "defaultHandler" to the exception object.

An exception handler in a non-JADE application can return any of the return values listed in the table under "Creating an Exception Handler", earlier in this chapter. The effects are similar, but are restated in the following table to clarify their effect in this context.

The following table lists the C++ defines that are provided in the jomdefs.h file.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Define</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>E_CONTINUE</td>
<td>Processing continues as if no error had occurred. The JADE Object Manager Application Programming Interface (API) call that caused the exception ultimately returns a zero result to the user application.</td>
</tr>
<tr>
<td>1</td>
<td>E_ABORT_ACTION</td>
<td>Aborts the current action. The JADE Object Manager message stack is cut back, and control returns to the user application, with the JADE Object Manager API call that caused the exception returning a zero result.</td>
</tr>
<tr>
<td>2</td>
<td>E_RESUME_NEXT</td>
<td>If the exception handler was armed in an external method, the JADE Object Manager message stack is cut back to this method and the JADE Object Manager API call that caused the exception returns a zero result. If there were no JADE Object Manager messages on the stack when the handler was armed, the effect of the E_RESUME_NEXT define is identical to that of the E_ABORT_ACTION call.</td>
</tr>
<tr>
<td>-1</td>
<td>E_PASS_BACK</td>
<td>Passes control back to any previously armed exception handler for this type of exception. If there are no other applicable handlers, the JADE Object Manager message stack is cut back and control returns to the user application, with the JADE Object Manager API call that caused the exception returning a non-zero result representing the error condition.</td>
</tr>
</tbody>
</table>

Using the raise_ Method

To raise user exceptions, use the raise_ method in the ExceptionClass class. (ExceptionClass is a subclass of the Class class in the Type parent class.) The raise_ method has the following syntax.

```java
exception-class-name.raise_(errorCode: Integer;
  errorItem: String);
```

The raise_ method automatically creates an exception of type exception-class-name and assigns the specified errorCode and errorItem properties before the exception is issued.

The following example shows a method that raises an exception of type NormalException with an error code of 55 and a description of "Invalid Company".

```java
initCompany(newCompany: Company io);
begin
  if newCompany = null then
    NormalException.raise_(55, "Invalid Company");
end;
```
... // perform the rest of the initialization logic
end;

The following example shows a method used to raise an exception of type GraphException when two arrays contain a different number of entries.

```pascal
checkForErrors(array1: StringArray; array2: IntegerArray);
begin
  if array1.size <> array2.size then
    GraphException.raise_(3, "Arrays are the wrong size");
  end;
end;
```

The GraphException class in this example is a user-defined Exception subclass created for the application.

### Using the `raise exception` Instruction

Use the JADE `raise` exception instruction to raise an exception using a previously created exception object. This enables you to record extra information with the exception before raising it.

The optional internal or precondition keyword specifies the type of exception, which is precondition by default. You can use the optional internal or precondition keyword of the raise exception instruction to specify whether an exception is an internal error or a precondition violation on the raise instruction. The default type is precondition. A precondition exception is used to report precondition contract violations where the caller has failed to satisfy precondition requirements of the invoked method.

When a method raises a precondition violation, it signifies that its caller has failed to meet certain requirements such as passing valid parameters or ensuring a required state in objects used by the invoked method for the called method to do its job.

When a precondition exception is raised at run time, the exception object contains two method descriptor references: currentMethodDesc, which describes the calling method, and the reportingMethodDesc, which describes the reporting method (that is, the method that raised the exception).

When an internal exception is raised, currentMethodDesc describes the method raising the exception and reportingMethodDesc contains a null reference. The default exception handler and dialog display information from the current method descriptor and the reporting method descriptor, if present.

The following example shows the method that creates an object of the UserException class, defines the properties for this object, and then raises the exception. (See also "Creating an Exception Handler" and "Examples of Arming an Exception Handler", earlier in this chapter.)

```pascal
method4();
vars
  userException : UserException;
begin
  create userException;
  userException.errorCode   := 64000;
  userException.continuable := true;
  userException.resumable   := true;
  raise userException;
  status.caption := "Resuming execution after raising exception";
end;
```

The following example shows a method used to raise an exception of type MyException.

```pascal
raiseMyException(code: Integer; categ: Integer);
vars
```
```pascal
exObj : MyException;
begin
  create exObj;
  exObj.errorCode := code;
  exObj.category := categ;
  raise exObj;
end;
```

The following method is an example of raising an exception for a variable exception class. The primitive type or class of the exception is passed as a parameter to the method.

```pascal
myRaiseException(exClass: ExceptionClass; error: Integer; text: String);
vars
  exObj : Exception;
begin
  create exObj as exClass;
  exObj.errorCode := error;
  exObj.extendedErrorText := text;
  raise exObj;
end;
```

## Interpreting Exceptions

Each **Exception** class provides the properties listed in the following table, to enable you to interpret exceptions.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>category</td>
<td>Integer</td>
<td>Categorizes exceptions within the <strong>UserException</strong> class</td>
</tr>
<tr>
<td>continuable</td>
<td>Boolean</td>
<td>Specifies if execution can be continued after the exception has been handled</td>
</tr>
<tr>
<td>currentMethodDesc</td>
<td>MethodCallDesc</td>
<td>Reference to a <strong>MethodCallDesc</strong> object describing the method that caused the exception</td>
</tr>
<tr>
<td>errorCode</td>
<td>Integer</td>
<td>Uniquely identifies the exception</td>
</tr>
<tr>
<td>errorItem</td>
<td>String</td>
<td>Additional information about the exception and displayed in the default exception dialog as <strong>error item</strong></td>
</tr>
<tr>
<td>extendedErrorText</td>
<td>String</td>
<td>Extended error description for exception instances recorded by any service that raises exceptions</td>
</tr>
<tr>
<td>helpBook</td>
<td>String</td>
<td>Windows help file containing the explanation of the exception and opened when you click the <strong>Help</strong> button in the default exception dialog</td>
</tr>
<tr>
<td>kind</td>
<td>Character</td>
<td>Specifies the kind of exception that was raised (raised by the caller or the receiver)</td>
</tr>
<tr>
<td>level</td>
<td>Integer</td>
<td>Level number of the exception and automatically incremented for each nested exception</td>
</tr>
<tr>
<td>remoteErrorCode</td>
<td>Integer</td>
<td>Identifies an exception that occurred while executing a method on another node</td>
</tr>
<tr>
<td>reportingMethodDesc</td>
<td>MethodCallDesc</td>
<td>Reference to a <strong>MethodCallDesc</strong> instance that describes the method that reported or raised the exception</td>
</tr>
</tbody>
</table>
The methods provided by the `Exception` class are listed in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createSOAPMessage</code></td>
<td>Returns a string representing a SOAP fault message</td>
</tr>
<tr>
<td><code>debug</code></td>
<td>Displays current process stack information and enables you to inspect variables</td>
</tr>
<tr>
<td><code>defaultHandler</code></td>
<td>Calls the <code>showDialog</code> method to display the default exception dialog</td>
</tr>
<tr>
<td><code>errorObject</code></td>
<td>Returns the object reference in error</td>
</tr>
<tr>
<td><code>initializationHandler</code></td>
<td>Placeholder for a user-defined exception handler for use during the JADE initialization process</td>
</tr>
<tr>
<td><code>logExceptionHistory</code></td>
<td>Logs the exception stack history of each nested exception</td>
</tr>
<tr>
<td><code>logProcessHistory</code></td>
<td>Logs the call stack history</td>
</tr>
<tr>
<td><code>logSelf</code></td>
<td>Appends a description of the exception object to a file</td>
</tr>
<tr>
<td><code>setErrorObject</code></td>
<td>Saves a reference to the error object in the exception instance</td>
</tr>
<tr>
<td><code>showDialog</code></td>
<td>Displays the default exception dialog</td>
</tr>
<tr>
<td><code>text</code></td>
<td>Returns the error text from the <code>jadmsgs.eng</code> file or other language file that corresponds to the <code>errorCode</code> number</td>
</tr>
</tbody>
</table>

The `ConnectionException` class defines the behavior of exceptions that occur as a result of connecting to external systems and provides the properties listed in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connection</code></td>
<td><code>Connection</code></td>
<td>Contains a reference to the connection object that caused the exception</td>
</tr>
<tr>
<td><code>dataBuffer</code></td>
<td><code>Binary</code></td>
<td>Contains the data that the user was trying to send when the exception was raised</td>
</tr>
</tbody>
</table>

The `DeadlockException` class defines the behavior of exceptions that occur as a result of deadlocks and provides the properties listed in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lockDuration</code></td>
<td><code>Integer</code></td>
<td>Contains the duration of the lock</td>
</tr>
<tr>
<td><code>lockTimeout</code></td>
<td><code>Integer</code></td>
<td>Contains the timeout period of the lock</td>
</tr>
<tr>
<td><code>lockType</code></td>
<td><code>Integer</code></td>
<td>Contains the type of the lock</td>
</tr>
<tr>
<td><code>targetLockedBy</code></td>
<td><code>Process</code></td>
<td>Contains the process that locked the object</td>
</tr>
</tbody>
</table>
The **DeadlockException** class also provides the methods listed in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>lockTarget</td>
<td>The target object of the deadlock</td>
</tr>
<tr>
<td>obtainedLock</td>
<td>A reference to an object over which this process has obtained a lock</td>
</tr>
</tbody>
</table>

The **FileNotFoundException** class defines the behavior of exceptions that occur as a result of file handling and provides the methods listed in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns an instance of the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>File or FileFolder class that was being used when the exception was raised</td>
</tr>
<tr>
<td>fileNode</td>
<td>FileNode class that was being used when the file exception was raised</td>
</tr>
</tbody>
</table>

The **LockException** class defines the behavior of exceptions raised as a result of locking conflicts, and defines the additional properties listed in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lockDuration</td>
<td>Integer</td>
<td>Corresponds to one of the lock durations in the LockDurations global constant category in the RootSchema</td>
</tr>
<tr>
<td>lockTimeout</td>
<td>Integer</td>
<td>Contains the timeout period of the lock</td>
</tr>
<tr>
<td>lockType</td>
<td>Integer</td>
<td>Corresponds to one of the lock types in the Locks global constant category in the RootSchema</td>
</tr>
<tr>
<td>retryCount</td>
<td>Integer</td>
<td>Contains the number of lock retries that were encountered</td>
</tr>
<tr>
<td>targetLockedBy</td>
<td>Process</td>
<td>Contains the process that locked the object</td>
</tr>
</tbody>
</table>

The methods provided by the **LockException** class are listed in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lockTarget</td>
<td>Returns the target object of the lock</td>
</tr>
<tr>
<td>retryLock</td>
<td>Retries the lock in a multiuser application, increments the retry count, and returns true if the lock was obtained</td>
</tr>
<tr>
<td>showDialog</td>
<td>Displays the default lock exception dialog</td>
</tr>
</tbody>
</table>

The **NotificationException** class defines the behavior of exceptions that occur as a result of notification events when the subscriber cannot be found and provides the method listed in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>notificationTarget</td>
<td>Returns the reference to the target for the notification whose subscriber was not found</td>
</tr>
</tbody>
</table>
The **ODBCException** class defines the behavior of exceptions that occur as a result of ODBC communications and provides the properties listed in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nativeError</td>
<td>Integer</td>
<td>Contains the native data-source-specific error code</td>
</tr>
<tr>
<td>state</td>
<td>String</td>
<td>Five-character ODBC-defined state variable</td>
</tr>
</tbody>
</table>

The **ODBCException** class also provides the method listed in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>showDialog</td>
<td>Displays the default exception dialog</td>
</tr>
</tbody>
</table>

---

**Monitoring Forms to Identify an Exception Dialog**

When you run a JADE application, a Microsoft Windows **Window** class is created for each kind of window to establish its basic operating style. This class is not visible to you unless you use a Windows or user-written utility to monitor the style of forms on the desktops of users.

The **Window** classes for forms in a JADE application have the following names:

- **Jade:Form** for a standard JADE form
- **JadeMsgBox:Form** for a message box created using the **msgBox** method of the **Application** class
- **JadeException:Form** for an exception dialog; that is, the standard exception, lock exception, lock retry, or ODBC exception dialog
Chapter 4  Collections Behavior and Tuning

This chapter covers the following topics.

- **Overview**
  - Tuning the Btree Load Factor
  - Tuning the Block Size at Collection Type Level
  - Tuning the Block Size at Instance Level
- **Collection Behavior**
  - Precedence of Block Size and Load Factor Values
- Tuning Collections at Development Time
- Using Methods to Dynamically Tune Collections at Run Time
- Analyzing a Collection
- Performance Considerations

**Overview**

You can tune the size of collection blocks and the load factor attribute at the collection type or instance level, by using the Tuning sheet on the Define Class dialog for a Collection subclass.

The JADE development environment enables you to tune collection block parameters at the type (collection class) level.

You can specify the collection block size and load factor attributes for a specific instance by using collection runtime instance methods.

The ability to tune the size of blocks is particularly useful in Btree structures, as the block size can determine the number of levels or the height of a Btree required to contain a specific population (number of entries). Minimizing the number of levels in a Btree has been shown to improve the overall performance for typical use cases with large collections.

The load factor attribute enables you to affect the loading of blocks when a Btree block split occurs. The optimal value for this is dependent on the key order in which entries are inserted; for example, purely sequential versus random insertion.

**Note** In this chapter, block size tuning applies to all collection types and load factor tuning is specific to Btree types (that is, sets and dictionaries).

For details about the behavior of collections, see "Collection Behavior", later in this chapter.

**Tuning the Btree Load Factor**

The load factor specifies the ratio of entries (as a percentage factor) that are moved to a new block when a Btree block splits.

However, as sets are intrinsically unordered, usually a 66 percent (random) load factor is appropriate.
For dictionaries, statistically a higher load factor value (for example, 95 percent) provides a denser loading of blocks when entries are added in sequential key order, or highly unbalanced Btrees may result.

**Tuning the Block Size at Collection Type Level**

You can specify the block size of a collection at the class level in one of the following ways.

- Indirectly, by providing an estimate of the expected population of the collection
- Directly, by specifying the block size in terms of entries in each block

For details, see "Tuning Collection Classes" under "Defining a Class", in Chapter 3 of the JADE Development Environment User's Guide. See also "Tuning Collections at Development Time", later in this chapter.

The indirect option provides a simple way for you to indicate that JADE should use a block size determined heuristically. However, as a heuristic-based computation is not guaranteed to come up with an optimal solution, you can specify the number of entries in each block explicitly if you have the appropriate information to do so.

These options are mutually exclusive. If both are defined, the explicit block size takes precedence.

**Tuning the Block Size at Instance Level**

Tuning a block size at instance-level by using the `Collection::setSize` method enables you to tune collections at run time. You could use this, for example, to support installation-specific customization where data volumes may vary between end-user sites.

**Collection Behavior**

Collections are composite objects that comprise a collection header and an aggregate of variable size collection blocks.

When you do not specify the collection block size, collections behave as follows.

- For any collection, the first block is instantiated with a size sufficient to contain exactly four entries. As entries are added to the collection, the collection block grows in increments up to the defined maximum size for the collection.
- The **List** and **Btree** collection types are comprised of collection blocks of varying sizes.
- **Array** blocks start with an initial size of four entries and grow to the maximum block size limit, as required. Random insertions into the middle of an array can result in a list structure comprising a range of block sizes.
- When a **Btree** root block has grown to the maximum number of entries for each block, the root block is split and all further blocks from then on are created with an actual size of the defined maximum size for the collection.

**Precedence of Block Size and Load Factor Values**

You can specify block sizes and load factor values at class or instance level. For details, see the following subsections.

**Class-Level Specification**

When you specify the block size for a collection class, the block size determines the default collection block size for instances of the collection type.
The specified block size becomes the actual size and collection blocks do not change in size during their lifetime. Similarly, when you specify the load factor for the collection class, the specified load factor is applied to instances of the collection class by default.

**Instance-Level Specification**

When you specify the block size for a collection instance, the specified block size overrides the class default.

The instance-level block size is the initial size and collection blocks do not change size during their lifetime. Similarly, when you specify the load factor for a collection instance, the specified load factor is applied to the instance, overriding the class default.

This behavior can be summarized by the rules of precedence for block sizes listed in the following table. **Class::MLB** represents the maximum logical block size defined for the collection class.

<table>
<thead>
<tr>
<th>Block Size at Instance Level (set dynamically at run time)</th>
<th>Block Size at Class Level (set in the Define Class dialog)</th>
<th>Collection Block Size</th>
<th>Size Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td>Zero (0)</td>
<td>b</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>Zero (0)</td>
<td>Zero (0)</td>
<td>4</td>
<td>Class::MLB</td>
</tr>
</tbody>
</table>

The rules of precedence for the load factor are listed in the following table.

<table>
<thead>
<tr>
<th>Instance-Level Load Factor</th>
<th>Class-Level Load Factor</th>
<th>Collection Load Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>n</td>
<td>m</td>
</tr>
<tr>
<td>Zero (0)</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Zero (0)</td>
<td>Zero (0)</td>
<td>Default (66 percent)</td>
</tr>
</tbody>
</table>

**Tuning Collections at Development Time**

You can use the Tuning sheet of the Define Class dialog to specify the block size and load pattern that you require for all instances of your Collection subclass at run time. You can select:

- The expected population of the collection block or the block size in units of number of entries per block
- A random (default) load pattern for Btree subclasses

For details, see "Tuning Collection Classes" under "Defining a Class", in Chapter 3 of the JADE Development Environment User’s Guide.

**Using Methods to Dynamically Tune Collections at Run Time**

JADE provides methods that enable you to dynamically specify the block size and load factor for individual collection instances at run time, as follows.

- The Collection class **setBlockSize** method enables you to specify or change the block size of the receiver.
- The Btree class **setLoadFactor** method enables you to modify the default load factor for a Btree-based collection (that is, for sets and dictionaries).
These methods provide maximum flexibility and support runtime tailoring based on expected volumes. For details, see Chapter 1 of the JADE Encyclopaedia of Classes.

Analyzing a Collection

You can use the Collection class getStatistics method to analyze the collection receiver and return structural statistics of the collections, as follows.

- Actual block size of the collection
- Size of the key in bytes (object id (6) for sets and Integer (4) for arrays)
- Size of each collection entry in bytes
- Total number of blocks in the collection
- Total number of entries in the collection
- Height (that is, the number of levels) in the collection (which is always 1 for arrays)
- Minimum number of entries found in any block
- Maximum number of entries found in any block
- Average number of entries in collection blocks
- The actual average percentage loading of collection blocks (entries for each block)

For details, see Chapter 1 of the JADE Encyclopaedia of Classes.

Performance Considerations

You can change the block size used in a Btree to affect the height or number of levels in a Btree collection.

Theory and test show that minimizing the height of a Btree reduces both the search and update time for Btree operations.

The improvements occur in the following areas.

- Reduced CPU time for Btree operations when Btree is cache-resident
- Improvements in overall disk and network retrieval times, by reducing the number of fetches, mitigating the effects of transfer latency

However, you should consider the following potential negative effects.

- Increase in transaction journal sizes, because the entire collection block images are audited
- Increased object cache utilization for certain collection access patterns
Chapter 5 Creating Your Own Control Classes

This chapter covers the following topics.

- Overview
- Subclassing a JADE Control Class
  - Selecting Your Design Time Properties
- Adding Properties to Your Subclassed Control
- Adding Methods to Your Subclassed Control
  - Examples

Overview

You can subclass a JADE control class to provide a new control to meet your own requirements.

Defining your own controls provides:

- Extended behavior of the Control class and subclasses provided by JADE; for example, you can define a text box that toggles foreground and background colors when it has focus.
- Specific behavior not provided by JADE system controls; for example, you can define an indicator used to represent an on/off situation, by subclassing the BaseControl class and then adding new properties and methods.

If you reimplement an event method, you must include an inheritMethod instruction in your code so that the corresponding event method in the standard control class is invoked.

The steps that you perform to create your own controls are:

1. Subclass a JADE control class
2. Add properties to your new control class
3. Add methods to your new control class

For details, see the following sections.

Notes

Abstract control classes (for example, the BaseControl and ScrollBar classes) are not displayed on the Control palette in the JADE Painter. However, an icon is added to the palette for the HScroll and VScroll subclasses of the abstract ScrollBar class and when you subclass the BaseControl class unless you uncheck the Can be shown in the Painter's control palette check box on the Options sheet of the Define Class dialog.

If you have mapping logic on subclassed controls, other processes such as the JADE Painter, Translator utility, or the loading of schemas may also execute that logic. The logic therefore may need to perform checks to determine if it is running in the user application environment, to ensure that exceptions are not generated in these other situations.

Subclassing a JADE Control Class

The first step in creating your own JADE control is to subclass an existing JADE control class.
To subclass a JADE control class

1. From the Class List of the Class Browser, select the control class that you want to subclass.
2. Select the Add command from the Classes menu. The Define Class dialog is then displayed.
3. Specify the name of your control in the Name text box of the Class sheet and then use the Subclass of combo box to select the control that you want to subclass; for example, BaseControl.
4. Select your map file in the Map File combo box and then select your access, type, and persistence options, if required.
5. Click the Options sheet. The Options sheet of the Define Class dialog is then displayed, to enable you to set properties that are specific to your class.

An example of the Options sheet is shown in the following image.

![Options sheet example](image)

6. Click the Options sheet so that you can:
   a. Uncheck the Can be shown in the Painter’s control palette check box if you do not want the image (icon) for your control displayed in the control palette of JADE Painter. (By default, the control is displayed.)
   b. The Painter Icon Definition group box enables you to select how the picture is to be displayed on the tool of the Painter Control palette for this control for each of the four states associated with them; that is, up, down, roll over, and disabled.

   Click the Up, Rollover, Down, or Disabled option button and then click the Change button to specify a toolbar icon or to change an existing icon for the selected image state (which defaults to the Up state). The common File Open dialog (titled Select the <state> image file name) is then displayed, to enable you to select an existing graphics file whose image you want displayed for the Control subclass in the selected state.
c. Repeat step b for each of the states associated with the image that you require.

Tip  Although you do not have to define images for each state associated with the control picture, a down state image provides users with visual confirmation that the icon has been clicked.

d. Click the Design Time Properties button if you want to specify the control properties that are listed in the Properties dialog during your Painter sessions. (A value assigned to a design time property can be saved and used as the default value when a runtime instance of the control is created.)

The Design Time Properties dialog is then displayed. For details about this dialog, see "Selecting Your Design Time Properties", in the following section.

When you have created your new control class (and specified your design time properties, if required), it is displayed as a subclass of the selected control class in the Class List of the Class Browser.

Selecting Your Design Time Properties

The Selection sheet of the Design Time Properties dialog is displayed when you click the Design Time Properties button in the Options sheet of the Define Class dialog, to enable you to select the properties that are listed in the Properties dialog during your Painter sessions. For example, if you want the design time properties for your control to differ from those of the parent class, use this dialog to specify the design time properties relevant to your new control.

You can specify that a property is a design time property of one class but not a design time property of a subclass. Additionally, the list of options for a property can differ between its subclasses and superclasses.

For a newly defined control, the Selected list box of the Selection sheet lists all properties of the parent control and its superclasses.

An example of the Selection sheet of the Design Time Properties dialog is shown in the following image.
When your control has been defined, any properties that you previously selected as design time properties for that control are listed in the **Selected** list box and disabled in the **Properties** list box.

**To select your design time properties**

1. In the **Properties** list box, select the property of the control superclass that you want to include in the Properties dialog during Painter sessions and then click the > button to copy the selected property to the **Selected** list box. (The selected property is then disabled in the **Properties** list box.)

   **Tip**  
   Double-click a property to quickly copy it from one list box to the other.

2. In the **Selected** list box, select the property that you do not want to include in the Properties dialog during Painter sessions and then click the < button to move the selected property to the **Properties** list box.

3. Repeat steps 1 and 2 for each property that you want to select or deselect as a design time property for your control.

4. Check the **Show Inherited** check box if you want the **Properties** list box to display the properties inherited from parent classes. (By default, inherited properties are not displayed.)

5. Select the **Input Type** sheet, to specify how your selected properties are displayed in the Properties dialog in Painter sessions (for example, to fine-tune the type or range of data in the type library of and imported control).

   Alternatively, click the **OK** button to confirm your selections and return focus to the Define Class dialog, or the **Cancel** button to abandon your selections.

**Controlling the Display in the Properties Dialog**

When you have selected the design time properties for your control and then clicked the **Input Type** tab, the **Input Type** sheet, shown in the following image, is then displayed.
Use the Input Type sheet of the Design Time Properties dialog to specify how your selected properties are displayed in the Properties dialog during Painter sessions; for example, a color selection dialog or a combo box that allows selection from a set of values.

If input is to be by a custom-built dialog, or property page, set the input type to Via Property Page in the Type combo box. (See also the getPropertyDisplay, hasPropertyPage, and showPropertyPage methods, in Chapter 2 of the JADE Encyclopaedia of Classes.)

For newly selected properties, the input type defaults to the most appropriate type; for example, a Binary primitive type property is given a default type of Picture. However, most are set to String, which is displayed in the Properties dialog as a text box that allows any string value to be specified.

To specify the input type of a property

1. Select the appropriate property in the Properties list box.

   The input type of that property is then displayed in the Type combo box.

2. In the Type combo box, select the input type that you require for the property in the Properties dialog; for example, an OLE object or a picture.

   **Note** JADE does not check that the selected input type is consistent with the property type; for example, you can select an Integer property for a String primitive type.

3. If the selected input type is List, use the Value text box to individually specify the values for that property. Each value that you specify for the property is then displayed in the List values list box, to enable you to select the values that you want to display in the Properties dialog.

4. To add a value to the selected design time property, specify the required value in the Value text box and then click the Add button to add your specified value to the List values list box. For details, see "Defining a List Value", in the following subsection.

5. To change a value, select the value in the List values list box and then change it to the appropriate value when it is then displayed in the Value text box. Click the Change button to amend the listed value.

   When a selected value is displayed in the Value text box and the text box has focus, click the Remove button to remove the value.

6. Use the up or down arrow button to change the order of the listed values; for example, to display the second value after the fifth value, select the second value and then click the down arrow button three times.

7. When you have finished selecting your design time properties and specifying the property input types, click the OK button. Focus is then returned to the Define Class dialog.

   Alternatively, click the Cancel button to abandon your selections.

**Defining a List Value**

The format of a List type is a value, optionally followed by a separator and a description; for example, 12 - No Drop. All JADE List values have the following format.

```
value [separator] [description]
```

The separator value is space, en dash symbol (-), space; that is, -.

When a property is changed in the Properties dialog, the value portion is saved in the database. When the combo box is displayed in the Properties dialog, the saved value is also used to select the line from the value list.
The value is detached from the combo box list by using the `getNextToken` method of the `String` primitive type and then casting to match the property type; that is, the value is the list text up to the first `getNextToken` delimiter (for details, see "getNextToken", in Chapter 1 of the JADE Encyclopaedia of Primitive Types), followed by casting to the correct type. For example, for a new control with an input type `Character` property and a Properties dialog `List` type with list entries `Casual` and `Permanent`, the `getNextToken` method gets a value of `Casual` or `Permanent`, which is then cast to a `C` or `P` character that is stored in the database.

Adding Properties to Your Subclassed Control

You can add new properties to your control class as you can for any other class.

These properties can be marked for inclusion in the Properties dialog that is optionally displayed in the JADE Painter. (For details, see "Controlling the Display in the Properties Dialog", earlier in this chapter.)

Adding Methods to Your Subclassed Control

You can add new methods to your control class as you can for any other class. Some of the methods that you add are likely to be the reimplementation of superclass methods.

Controls have the following method types.

- Event external methods
- Standard methods

Event methods are methods that are usually triggered by an event.

Event methods execute the logic in the method and then call a method (whose method name is specified by the `control-name_this-method-name` signature) of the form on which the control is placed.

**Notes** When you reimplement an event method, include the `inheritMethod` call.

You must add event methods as external methods.

When you create your own external event methods, the highest method that is not reimplemented has the following format.

```java
event-name(parameter-list) is CallControlEvent in jadpmap updating;
```

At run time, it is this call that causes a form message to be generated. Methods are event methods only if the highest level of method implementation has a signature with this format.

When a control is displayed during painting (that is, at development time), the control can respond only to the `create`, `paint`, and `windowCreated` events.

**Note** If you reimplement these methods, ensure that you include an `inheritMethod` call. For example, if your subclassed control implements the `click` method, calling the `inheritMethod` instruction causes the `control-name_click` method to also be executed. When you add an event to your control, JADE does nothing to that event unless you write code to implement it.

For details, see the appropriate method in Chapter 2 of the JADE Encyclopaedia of Classes.

Generally, where there is a requirement to set default values for properties, a constructor is implemented for the class. Alternatively, your method can perform initialization logic in the `windowCreated` event. You can also reimplement the following methods that control how your subclassed control is manipulated in the JADE Painter.
isMoveable
This method returns true if the control can be dragged to move it around the form in Painter.

isSelectable
This method returns true if the control can be selected in Painter.

**Tip** Reimplement this method in your application to return false if you have created a custom (user-defined) control that has other controls embedded in it. In most cases, you would not want the embedded controls to be selected during painting.

isSizeable
This method returns true if the control can be resized in Painter.

canBeChildOf
This method returns true if the JADE Painter allows the control to be placed on the specified form or control.

canControlHaveChildren
This method returns true if the JADE Painter allows a control to be dropped on top of this control.

canHaveAsChild
This method returns true if the JADE Painter allows the specified control to be placed on this control.

For details, see Chapter 2 of the *JADE Encyclopaedia of Classes*.

In the Properties dialog, you can control how the property value is displayed and the values that are acceptable (that is, you can provide validation of the input value). To do this, reimplement the following methods.

**Window::getPropertyDisplay**
This method gets the textual representation of the value to display.

**Window::hasPropertyPage**
This method returns true if the property has its own dialog for input and validation.

**Control::saveProperties**
This method saves the properties edited by the property page dialogs.

**Window::showPropertyPage**
This method displays a dialog that has previously been defined for the input and validation of the property value.

### Examples

The following example shows the type definition of an **Indicator** control that displays a caption, and is used to indicate an on/off condition. This control is a subclass of the **BaseControl** class and it has a value property that indicates whether the state of the control is on or off. This property has a mapping method that toggles foreground and background colors when the value property changes. It also reimplements the click and paint events.
The attributes defined for the Indicator class are listed in the following table.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Primitive Type</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>caption</td>
<td>String</td>
<td>1000</td>
</tr>
<tr>
<td>value</td>
<td>Boolean</td>
<td></td>
</tr>
</tbody>
</table>

The following is an example of a value mapping method.

```haskell
value(set: Boolean; val: Boolean io) mapping, updating;
vars
tempColor : Integer;
begin
  if set then
    if val <> value then
      // Swap the foreground and background colors to indicate
      // that the status has changed. This causes a paint.
      tempColor := backColor;
      backColor := foreColor;
      foreColor := tempColor;
    endif;
  endif;
end;
```

The following are examples of the reimplemented click and paint events.

```haskell
click(cntrl: Indicator) updating;
begin
  // The following assignment causes the value mapping method
  // to be executed.
  value := not value;
end;

paint(cntrl: Indicator input) updating;
begin
drawTextAlign := 2;
drawTextIn (caption, 0, 0, width, height, foreColor);
inheritMethod(cntrl);
end;
```

The following is an example of a constructor method.

```haskell
create() updating;
begin
  value := false;
end;
```

The following examples for an ActiveTextBox control (a subclass of the TextBox class) toggle foreground and background colors when the control has focus. The ActiveTextBox control has a toggleColors method that toggles foreground and background colors. It also reimplements the gotFocus and lostFocus event methods that call the toggleColors method.

```haskell
toggleColors() updating, protected;
vars
tempColor : Integer;
begin
tempColor := foreColor;
```
foreColor := backColor;
backColor := tempColor;
end;

gotFocus(textbox: ActiveTextBox input) updating;
begin
  toggleColors;
  inheritMethod(textbox);
end;

lostFocus(textbox: ActiveTextBox input) updating;
begin
  toggleColors;
  inheritMethod(textbox);
end;
Chapter 6  JADE Locking

This chapter covers the following topics.

- Overview
- Locking Objects
  - Exclusive Locks
  - Shared Locks
  - Reserve Locks
  - Update Locks
  - Cache Concurrency
  - Handling Deadlocks
  - Lock Handling by the JADE Server
  - Upgrading and Downgrading Locks
  - Lock Contention Information
- Unlocking Objects
  - Using Object Editions

Overview

The JADE Object Manager client maintains a local cache of objects. When a request is made for an object, the client checks its local cache before loading the object from the server. If the object is not present in the local cache, it is loaded from the server into the client local cache.

The object is the unit of transfer between the client and the server.

When an object is updated, a lock is automatically registered with the server if the object has not been manually locked in an exclusive manner.

When an object is locked, the client passes the edition of the object in local cache to the server. If the local copy is out of date, a new copy of the object is passed to the client.

If an object is to be locked, the first reference to it should be the lock operation itself.

For details about locking when reading and writing transactions and examples of transaction locks and applying shared locks on global objects, see also "Using Read and Write Transactions", in Chapter 1.

For a discussion about locking, see the Erewhon.pdf file that is included in the erewhon subdirectory of the examples directory on the JADE release medium.
You can optimize locking based on the frequency at which an object is updated, by specifying that an object has one of the following volatility states.

- **Volatile**, indicating that the object is often updated
  
  A volatile object is recorded as locked by any processes that have it locked until they unlock it, and whenever locked, the latest edition of the object is fetched from the database.

- **Stable**, indicating that the object is updated infrequently
  
  A shared lock in a stable object is locked by the process and by the node. A node lock is retained, avoiding a call the server on subsequent locks on that node, until the object is swapped out of cache or an exclusive lock is requested on any node.

- **Frozen**, indicating that the object is not updated
  
  A frozen object in cache is assumed to always be the latest edition, avoiding the fetch from the server node. A frozen object must have its volatility changed before it can be changed.

The **Object** class provides the cache concurrency methods summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>changeObjectVolatility</td>
<td>Changes the volatility of a persistent object</td>
</tr>
<tr>
<td>getObjectIdVolatility</td>
<td>Returns the volatility state of the specified object</td>
</tr>
<tr>
<td>isObjectFrozen</td>
<td>Determines whether the receiver object is frozen (not updated)</td>
</tr>
<tr>
<td>isObjectStable</td>
<td>Determines whether the receiver object is stable (updated infrequently)</td>
</tr>
<tr>
<td>isObjectVolatile</td>
<td>Determines whether the receiver object is volatile (often updated)</td>
</tr>
<tr>
<td>makeObjectFrozen</td>
<td>Makes the specified persistent object frozen</td>
</tr>
<tr>
<td>makeObjectStable</td>
<td>Makes the specified persistent object stable</td>
</tr>
<tr>
<td>makeObjectVolatile</td>
<td>Makes the specified persistent object volatile</td>
</tr>
</tbody>
</table>

Volatility applies only to persistent objects. All transient objects are considered volatile. For details, see "Cache Concurrency", later in this chapter.

The **Object** class provides methods that enable you to specify the action that is taken if a lock cannot be obtained on an object. For more details, see the following methods under "Object Class", in Volume 2 of the **JADE Encyclopaedia of Classes**.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>exclusiveLock</td>
<td>Attempts to acquire an exclusive lock on the specified object</td>
</tr>
<tr>
<td>getLockStatus</td>
<td>Gets the status of the specified lock</td>
</tr>
<tr>
<td>isLockedByMe</td>
<td>Returns <strong>true</strong> if you are the owner of the lock on the specified object</td>
</tr>
<tr>
<td>lock</td>
<td>Obtains a lock of the specified type or raises a lock exception if the lock cannot be obtained</td>
</tr>
<tr>
<td>reserveLock</td>
<td>Attempts to acquire a reserve lock on the specified object</td>
</tr>
<tr>
<td>sharedLock</td>
<td>Places a shared lock on the specified object</td>
</tr>
<tr>
<td>tryLock</td>
<td>Returns <strong>true</strong> if the lock was obtained or returns <strong>false</strong> if the lock could not be obtained (your code determines the action that is then taken)</td>
</tr>
</tbody>
</table>
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unlock</td>
<td>Removes the current lock from the specified object</td>
</tr>
<tr>
<td>updateLock</td>
<td>Attempts to acquire an update lock on the specified object</td>
</tr>
</tbody>
</table>

The **Node** class describes the behavior of the JADE node. For details, see the following method under "**Node Class**", in Volume 2 of the *JADE Encyclopaedia of Classes*.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLocks</td>
<td>Populates an array with transient instances of the current locks for the shared transient objects in the node</td>
</tr>
</tbody>
</table>

The **Process** class describes the behavior of the JADE process. For details, see the following methods under "**Process Class**", in Volume 2 of the *JADE Encyclopaedia of Classes*.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isCommitting</td>
<td>Returns true if the process is currently committing a transaction</td>
</tr>
<tr>
<td>setDefaultLockTimeout</td>
<td>Programatically changes the default lock timeout period for the receiving process</td>
</tr>
<tr>
<td>useUpdateLocks</td>
<td>Update locks rather than Exclusive locks are implicitly acquired when an object is updated</td>
</tr>
</tbody>
</table>

The **System** class describes the behavior of the JADE system. For details, see the following methods under "**System Class**", in Volume 2 of the *JADE Encyclopaedia of Classes*.

<table>
<thead>
<tr>
<th>Method</th>
<th>Populates ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>getLocks</td>
<td>An array with transient instances of the current locks held by all processes in the system</td>
</tr>
<tr>
<td>getObjectLockProcesses</td>
<td>A dictionary parameter with processes that have locks on the specified object</td>
</tr>
<tr>
<td>getQueuedLocks</td>
<td>An array with lock requests that are waiting for objects to be unlocked by the processes that currently have them locked</td>
</tr>
</tbody>
</table>

The **Lock** class describes the lock requests maintained by the system, and provides the **target** method that is used to get the object that is the target of a lock request.

The method summarized in the following table is defined in the **Lock** class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>Gets the object that is the target of the lock request</td>
</tr>
</tbody>
</table>

In addition, the **Lock** class provides the properties summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>Specifies when the object is automatically unlocked</td>
</tr>
<tr>
<td>elapsedTime</td>
<td>Contains the time that the lock has been in place</td>
</tr>
</tbody>
</table>
Property | Description
--- | ---
kind | Contains the type of lock (that is, normal, local, node, or node lock to be removed)
lockedBy | Contains the process that currently holds a lock
requestedBy | Contains the process that submitted the lock request
requestTime | Contains the date and time of the lock request
type | Contains the type of lock request
waitTime | Contains the length of time the lock request waits

For more details, see "Lock Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

The RootSchema provides two generic lock exception handler methods. You can arm these lock exception handlers in your own applications to handle lock exceptions. These can be armed as local exception handlers or as global exception handlers; for example:

```java
// global exception handler
on LockException do app.globalLockException(exception) global;

// local exception handler on the global object
on LockException do global.lockExceptionHandler(exception);
```

### Locking Objects

In the JADE language, you can use the exclusiveLock, sharedLock, reserveLock, and updateLock methods of the Object class to manually lock objects. (The JADE Application Programming Interface also provides calls for locking objects manually from external languages.) The valid concurrent lock combinations are shown in the following table.

<table>
<thead>
<tr>
<th>Share</th>
<th>Reserve</th>
<th>Update</th>
<th>Exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Reserve</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Update</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Exclusive</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

More than one shared lock can coexist for the object at any time but only one exclusive lock is allowed. Additionally, a reserve lock can coexist with any number of shared locks but is incompatible with another reserve lock or with an exclusive lock. If the object to be locked is already locked in an incompatible manner by another user or sub-user, an exception is raised.

The lock types and their global constant definitions (if applicable) are listed in the following table.

<table>
<thead>
<tr>
<th>Locks Global Constant</th>
<th>Integer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Get_Lock</td>
<td>0</td>
<td>Waiting to acquire a lock and forcing all other lock requests to be queued</td>
</tr>
<tr>
<td>Share_Lock</td>
<td>1</td>
<td>Shared lock.</td>
</tr>
<tr>
<td>Reserve_Lock</td>
<td>2</td>
<td>Reserve lock.</td>
</tr>
</tbody>
</table>
Locks Global Constant | Integer | Description
--- | --- | ---
Exclusive_Lock | 3 | Exclusive lock
Update_Lock | 4 | Update lock

The lock duration global constant definitions are listed in the following table.

LockDurations Global Constant | Integer
--- | ---
Transaction_Duration | 0
Session_Duration | 1
Persistent_Duration (reserved for future use) | 2

The lock timeout global constant definitions are listed in the following table.

LockTimeouts Global Constant | Integer Value
--- | ---
LockTimeout_Immediate | -1
LockTimeout_Infinite | Max_Integer (#7FFFFFFF)
LockTimeout_Process_DEFINED | -2 (use the process-defined default lock request timeout)
LockTimeout_Server_DEFINED | 0 (use the server-defined default)

You can specify a default lock timeout for a background process in the BackgroundProcessServerTimeout parameter in the [JadeClient] and [JadeServer] sections of the JADE initialization file. The background process lock timeout is specified in milliseconds, with the default value of 30000 (that is, 30 seconds).

**Caution** You should not change the BackgroundProcessServerTimeout parameter value unless the background process is having locking problems. Before you increase this value, examine the application to determine which objects are being locked and whether locks are being held for too long. For example, new nodes cannot sign on if the system.nodes dictionary is locked by the application. It is better to change the application to minimize the locking of system collections.

**Exclusive Locks**

Before an object can be updated, it must have an **exclusive** lock placed on it. When you lock an object using an **exclusive** lock, no other process can lock the same object. JADE automatically applies an **exclusive** lock when an object is updated. By default, updated objects are locked for the duration of the transaction.

**Shared Locks**

If you lock an object using a **shared** lock, other processes attempting to update the object or explicitly acquire an **exclusive** lock wait until the lock is released but can acquire a **shared** lock or a **reserve** lock.

**Reserve Locks**

A **reserve** lock is available for situations where you intend to update an object but you need to minimize the length of time the object is locked with an **exclusive** lock.
When you place a **reserve** lock on an object, other processes attempting to acquire an **exclusive** lock or a **reserve** lock on that same object wait until the **reserve** lock is relinquished, but those processes attempting to acquire a shared lock succeed.

**Update Locks**

An **update** lock allows updates but is compatible with **shared** locks; that is, an **update** lock allows a single writer and multiple readers, because of transaction isolation.

The **update** lock has strength between a **reserve** and an **exclusive** lock. Whereas an **exclusive** lock blocks **shared** locks, an **update** lock allows updates but is compatible with **shared** locks; that is, an **update** lock allows a single writer and multiple readers, because of transaction isolation.

**Update** locks reduce locking contention in JADE systems; for example, bottlenecks involving persistent collections in systems with a large number of transactions.

The **update** lock allows an object to be updated but without blocking other processes obtaining **shared** locks. The process with an **update** lock can update the object. Other processes can acquire **shared** locks on the object, but they see the most-recent committed edition.

**Update** locks can be automatically enabled by using the **Process** class **useUpdateLocks** method, as shown in the following example.

```java
process.useUpdateLocks(true);
```

The automatic lock applied when an object is first updated is an **update** lock rather than an **exclusive** lock. Conversely, set the parameter to **false** to disable the use of the **update** locks.

Before the process commits the transaction, **update** locks are automatically upgraded to **exclusive** locks, by blocking all subsequent **shared** locks and then upgrading each **update** lock. (An **update** lock cannot be upgraded until all existing **shared** locks are released.)

When an **update** lock is being placed and the process already has a **shared** lock on the object, the **shared** lock is released before the **update** lock is requested. The release of the lock occurs despite the process being in transaction state. The lock is not released if the **shared** lock is of **session** duration. This prevents potential **shared** lock deadlocks, but allows the possibility of another process updating the object between the releasing of the **shared** lock and the acquiring of the **update** lock. If this happens, a continuable exception is raised.

If releasing the **shared** lock is going to be a problem for your application, the process can manually acquire the **update** lock before using the object; that is, before the **shared** lock would otherwise be acquired.

**Exception Handling**

Because **update** locks are upgraded to **exclusive** locks before committing transactions, object locked and deadlock exceptions can occur.

Object locked exceptions are reported as normal lock exceptions (that is, error code 1027), which allows exception handlers to determine the object involved, to retry the lock, and to abort or continue the lock, as required. You should retry an **exclusive** lock for the object involved.

When a lock exception is continued, the commit carries on, upgrading the remaining **update** locks to **exclusive** locks and then committing the updates.

If the lock exception handler attempts to continue without successfully retrying the lock, an exception is raised. If the lock exception is not successfully continued, the transaction is automatically aborted.

Use the **Process** class **isCommitting** method to determine if a process is currently committing. This method returns **true** if the process is currently committing a transaction.
Deadlock exceptions during commit are normal deadlock exceptions (that is, error code 1081), with standard exception information available.

When the lock on an object is upgraded from shared to update, the object is unlocked before the update lock is requested. If a different process updates the object before the update lock is acquired, exception 1146 (The object was updated before the lock upgrade completed) is raised.

If the object were to be updated, an exception is raised. If it does not matter to the application that the object was updated before the update lock could be acquired, the exception handler can effectively ignore the exception by returning Ex_Continue, allowing the commit to carry on and take place.

## Cache Concurrency

JADE enables you to optimize locking based on the frequency at which an object is updated, by specifying that an object has one of the following categories.

- **Volatile**, indicating that the object is often updated (the default)
  
  Objects are volatile by default. Volatile objects are locked and unlocked in the usual manner; that is, a volatile object is recorded as locked by any processes that have it locked until they unlock it, and whenever locked, the latest edition of the object is fetched from the database.

- **Stable**, indicating that the object is updated infrequently
  
  The infrequency of update allows an optimization to avoid having to request the JADE database server to fetch the latest edition of an object from the database every time an object is shared locked. Instead, when a shared lock on a stable object is first obtained on a specific node, as well as being locked by the requesting process it is deemed locked by that node.

  This "node lock" remains in place even after the process unlocks the object so that the next time a process requests a shared lock on that object, there is no need to fetch the object from the database if the node lock is still in place. If no processes have the stable object shared locked, the node lock remains in place until the object is swapped out of cache or a process on any node requests an exclusive lock on the object. For details, see "Stable Objects", later in this chapter.

- **Frozen**, indicating that the object is not updated
  
  The frozen volatility state allows an optimization that assumes a frozen object in cache is always the latest edition and that there is no need to keep fetching the object from the server node or indeed to record the object as locked in response to shared lock requests.

  As frozen objects cannot be updated, you must first change the volatility of a frozen object. You should do this with extreme caution, as other processes using the object do not necessarily know that it has been updated, even though those processes may have the object shared locked. For details, see "Frozen Objects", later in this chapter.

An example of where the volatility feature can provide benefit is when iterating through collections, using iterators or the foreach instruction. Making a collection stable or frozen improves performance, as there is no need to request the server to fetch the collection header object from the database every time the collection is to be iterated in response to the implicit shared lock requests associated with iterating collections.

**Note** Frozen and Stable volatility applies to persistent objects only. All transient objects are considered Volatile.
The **ObjectVolatility** category global constants for the object volatility states are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volatility_Frozen</td>
<td>#04</td>
<td>Object is frozen (that is, it is not updated)</td>
</tr>
<tr>
<td>Volatility_Stable</td>
<td>#08</td>
<td>Object is stable (that is, it is updated infrequently)</td>
</tr>
<tr>
<td>Volatility_Volatile</td>
<td>#00</td>
<td>Object is volatile (that is, it is updated often)</td>
</tr>
</tbody>
</table>

### Defining Volatility State

You can specify volatility for specific objects or for all instances of a specific class.

The **Object** class methods summarized in the following table enable you to define the volatility state of a specific object. (For details, see Volume 2 of the *JADE Encyclopaedia of Classes.*)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>changeObjectVolatility</td>
<td>Changes the volatility of a persistent object</td>
</tr>
<tr>
<td>makeObjectFrozen</td>
<td>Makes the specified persistent object frozen</td>
</tr>
<tr>
<td>makeObjectStable</td>
<td>Makes the specified persistent object stable</td>
</tr>
<tr>
<td>makeObjectVolatile</td>
<td>Makes the specified persistent object volatile</td>
</tr>
</tbody>
</table>

A conditional frozen object can be updated only by first changing its volatility to volatile or stable.

The Define Class dialog provides the **Volatility** sheet, which enables you to maintain the volatility of all instances of a class. (By default, class instances are volatile.) For details, see "Maintaining Class Instance Volatility", in Chapter 3 of the *JADE Development Environment User’s Guide.*

### Determining Object Volatility

The **Object** class methods summarized in the following table enable you to determine the volatility state of a specific object. (For details, see Volume 2 of the *JADE Encyclopaedia of Classes.*)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getObjectVolatility</td>
<td>Returns the volatility state of the specified object</td>
</tr>
<tr>
<td>isObjectFrozen</td>
<td>Determines whether the receiver object is frozen (not updated)</td>
</tr>
<tr>
<td>isObjectStable</td>
<td>Determines whether the receiver object is stable (updated infrequently)</td>
</tr>
<tr>
<td>isObjectVolatile</td>
<td>Determines whether the receiver object is volatile (often updated)</td>
</tr>
</tbody>
</table>

### Stable Objects

Acquiring shared locks for stable objects incurs much less overhead, at the expense of extra cost when acquiring an exclusive lock. However, as stable objects are updated infrequently, exclusive locks are not often required.

A stable object is deemed locked at a **node** level as well as individually by **process**, and the lock at node level remains in place as a dormant lock even when processes no longer have the object locked. Instead of being released immediately the object is unlocked, a node lock remains in place until required to be released.

You can use the read-only **kind** property of the **Lock** class to distinguish node locks.
The constants provided by the Lock class and contained in the kind property are listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Character Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind_Local</td>
<td>'01'</td>
<td>Applies to stable objects and represents a shared, transient duration lock that has an associated node lock (Kind_Node) entry in the database server lock tables. There is no individual lock entry for the process in the database server lock tables (unless the process is a server application).</td>
</tr>
<tr>
<td>Kind_Node</td>
<td>'02'</td>
<td>Applies to stable objects and represents a shared, transient duration lock that may be held by one or more processes on the node associated with the node lock. The associated background process of the node is used as the locking process. A node lock is released when an exclusive lock request is received or the object is removed from the node's cache and there are no processes on the associated node that have the object locked with local locks.</td>
</tr>
<tr>
<td>Kind Normal</td>
<td>'00'</td>
<td>Represents a lock held by a process. It is released when the process unlocks the object.</td>
</tr>
</tbody>
</table>

The JADE Monitor displays node locks as shared locks, transaction duration, node kind, locked by the background process of the associated node.

**Shared Locks on Stable Objects**

When a stable object is shared locked by a process executing on a specific node, the local node records that the process has the object locked and that the node has the object locked. The database server lock tables record only the node lock.

When a process unlocks a stable object, the node lock remains in place. While the node lock is in place, any process on that node can acquire a shared lock for that object without having to send a request to the server node (provided that the object is still in the local memory cache), because the node lock ensures that the object is recorded as shared locked and therefore cannot have been updated.

A node lock is released when an exclusive lock request for the stable object is received or when it is removed from the local memory cache, provided no processes on that node currently have the object shared locked. If processes have the object shared locked, the node lock is released when the processes no longer have the object shared locked.

When examining locks that are in place on a system (for example, by using the JADE Monitor), a single node lock entry is displayed for a stable object that has been or that is currently shared locked on a client node. The individual processes that have the object shared locked are not displayed.

For node locks, the background process of the client node is always displayed as the process that has the object locked, even though it does not actually have it locked at the time. You can use the kind property of the Lock class to distinguish node locks.

**Reserve and Session Duration Locks on Stable Objects**

A node lock represents a shared, transaction duration lock.

Requests for reserve, exclusive, or session duration locks are handled normally; that is, the request is forwarded to the server node and the individual lock is recorded in the database server lock tables.
Exclusive Locks on Stable Objects

An exclusive lock on a stable object cannot be acquired until all node locks (representing shared locks) are released. Because of this, all nodes that have node locks in place for the object are requested to release the node locks when an exclusive lock is requested. This is done at the earliest opportunity: a node releases a node lock immediately if there are no processes that have the object shared locked or it is released as soon as there are no processes remaining on that node with the object shared locked.

When an exclusive lock request on a stable object is received, all other lock requests on that object are queued. This is true even if the object is currently shared locked and another shared lock request is received. In this situation, all shared lock requests are forwarded to the server node, even if the node lock is still in place. This prevents the exclusive lock request being held up for long periods because of shared locks continuing to be acquired.

Server Applications, Single User Mode, and Server Execution Methods

The optimization for stable objects applies to:

- Applications running on client nodes
- Server applications
- Single user mode

However, the optimization is not carried out when locking stable objects within serverExecution methods; that is, serverExecution methods use normal locking procedures even if an object has a stable volatility state.

---

**Note**  When a stable object with a shared or reserve lock that was acquired in a serverExecution method is unlocked in a clientExecution method, the server node is not informed and the object remains locked on the server node. Frozen and volatile objects are unlocked on both nodes.

Frozen Objects

As frozen objects cannot be updated, there is no need to take any notice of shared or reserve locks, as objects that cannot be locked cannot become obsolete. Exclusive locks are still actioned, as they may be required for process coordination or synchronization.

The main benefit of using frozen objects is for iterating collections. The implicit shared lock requests associated with iterating are ignored, avoiding the overhead that would otherwise be involved.

Shared and Reserve Locks on Frozen Objects

For frozen objects, requests for shared or reserve locks are ignored. No locks are recorded.

Exclusive Locks on Frozen Objects

For frozen objects, exclusive lock requests are handled as normal; that is, the request is forwarded to the server node, is queued if the object is already exclusively locked, and is recorded when acquired.

**Note**  Because shared and reserve locks are ignored, exclusive locks can be acquired on frozen objects even if other processes have requested shared or reserve locks and have not yet unlocked them.

Updating Frozen Objects

As frozen objects cannot be updated, attempting to update a frozen object raises an exception.
When creating a persistent instance of a class that has the instance volatility set to Frozen, the object does not become frozen until it is committed. If an existing frozen object requires updating, you must change the volatility state, as shown in the following code fragment.

```java
beginTransaction;
    makeObjectVolatile(frozenObj);
    frozenObj.attr1 := 23;
    makeObjectFrozen(frozenObj);
commitTransaction;
```

Using these methods, the volatility state change from frozen to volatile is conditional; that is, if the object is being used by any other process, an exception is raised and the volatility is not altered. In a multiuser application where production mode is set, it is not possible to determine whether an object is in use by another process so an exception is always raised.

If you require an unconditional change (that is, the change takes place even if the frozen object is being used by another process), call the Object class changeObjectVolatility method, as shown in the following code fragment.

```java
beginTransaction;
    changeObjectVolatility(frozenObj, Volatility_Volatile, false);
    frozenObj.attr1 := 23;
    makeObjectFrozen(frozenObj);
commitTransaction;
```

**Caution** Take great care when making an unconditional change to the frozen volatility state, as the object can then be updated even if other processes have issued shared or reserve lock requests and not yet issued unlock requests. This may result in these processes accessing obsolete or inconsistent data, even though they have issued lock requests.

The safest approach when using an unconditional volatility change is to ensure that no applications that could use the object are active when the frozen object is to be updated.

### Freezing and Thawing Schema Files

The jdbutilb batch JADE Database utility provides the optional freezeSchemaFiles and thawSchemaFiles commands, enabling you to change the volatility of the _userint, _userscm, and _userxrf schema files to Frozen or to Transparent, respectively. The commands can be executed only when the database is offline.

Setting the file volatility to Frozen overrides any specified object volatility for all instances defined in that file, which means that all objects contained in a frozen schema file are themselves frozen.

Freezing schema files provides performance improvements, by reducing the number of locks, unlocks, and fetched user schema objects. This benefit results from ignoring implicit lock requests on objects stored in frozen schemas.

**Note** Attempts to create or maintain any development objects (for example, all schemas, classes, methods, forms, and so on) are disallowed while the schema files are in a frozen state. An attempt to modify any such object raises exception 1106 (Cannot update a frozen object).

This also applies to maintenance activities outside the JADE development environment, includingjadloadb batch schema loads. Before these maintenance activities can be carried out, the volatility of schema files must be set to Transparent by using the thawSchemaFiles command.
Handling Deadlocks

When a deadlock situation is detected between two processes, JADE provides flexibility in choosing which process should be given the deadlock exception. Processes have separate priorities for resolving deadlocks involving persistent objects and shared transient objects.

A process can set a "deadlock priority" value. When a deadlock occurs between two processes, the process that has the lower priority is given the deadlock exception. If the processes have identical priorities, the process that was requesting the lock is given the deadlock.

If double deadlock detection is enabled, both processes are given a deadlock exception, regardless of their relative priorities.

The deadlock priority value is a signed integer, with the default value being zero (0).

For details about the Process class getPersistentDeadlockPriority, getTransientDeadlockPriority, setPersistentDeadlockPriority, and setTransientDeadlockPriority methods, see Chapter 1 of the JADE Encyclopaedia of Classes.

See also the DoubleDeadlockException parameter in the [JadeServer] section of the JADE initialization file, in the JADE Initialization File Reference.

Lock Handling by the JADE Server

The JADE server arbitrates between multiple client processes making requests. The server coordinates multiple requests for locks and maintains details of locked objects based on lock requests placed by clients.

Upgrading and Downgrading Locks

The actions described in the following subsections apply when upgrading or downgrading locks.

Lock Durations

The lower-level lock durations are transaction locks and the higher-level durations are session locks.

 UPgrades

Lock duration upgrades are always attempted. The lock type is that of the highest type of the two locks. However, if the highest lock type is that of the shorter lock duration, the lock type is downgraded at the end of the transaction.

 DOWNgrades

The duration of a lock is never downgraded.

Lock Types

The lowest-level lock types are shared locks, followed by reserve locks, then update locks, and the highest-level lock types are exclusive locks.

 UPgrades

Lock type upgrades are always attempted. If the duration of the upgrading lock is shorter, the upgrade of the lock type will be for that duration only.

 DOWNgrades
Lock types are downgraded only when the duration of the new lock is greater than or equal to the duration of the existing lock. There are no lock type downgrades in transaction state. As lock type downgrades are delayed until after transaction state, only session locks survive after the end of a transaction for possible downgrading.

At the end of transaction state, all session locks are restored to the lock state to which they were last set manually, regardless of any lock type downgrade or upgrade done by transaction locks.

The following table lists lock upgrade and downgrade results.

<table>
<thead>
<tr>
<th>Upgrading Duration</th>
<th>Downgrading Type</th>
<th>Same Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attempted</td>
<td>Attempted at the end of transaction or load state</td>
<td>Attempted</td>
</tr>
<tr>
<td>Attempted for the upgrading lock type</td>
<td>Ignored</td>
<td>Ignored</td>
</tr>
<tr>
<td>Attempted</td>
<td>Attempted at the end of transaction or load state</td>
<td>Ignored</td>
</tr>
</tbody>
</table>

The following table lists examples of results when upgrading locks.

<table>
<thead>
<tr>
<th>Old Lock</th>
<th>New Lock</th>
<th>Upgrade Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transaction exclusive</td>
<td>Session exclusive</td>
<td>Session exclusive</td>
<td>Duration upgrade</td>
</tr>
<tr>
<td>Transaction shared</td>
<td>Session exclusive</td>
<td>Session exclusive</td>
<td>Duration and type upgrades</td>
</tr>
<tr>
<td>Transaction exclusive</td>
<td>Session shared</td>
<td>Session exclusive</td>
<td>Up to end of transaction, then automatically downgraded to session shared</td>
</tr>
</tbody>
</table>

The following table lists examples of results when downgrading locks.

<table>
<thead>
<tr>
<th>Old Lock</th>
<th>New Lock</th>
<th>Downgrade Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session shared</td>
<td>Transaction exclusive</td>
<td>Session exclusive</td>
<td>Up to end of transaction, then automatically downgraded to session shared</td>
</tr>
<tr>
<td>Session exclusive</td>
<td>Transaction exclusive</td>
<td>Session exclusive</td>
<td>No duration downgrade</td>
</tr>
<tr>
<td>Transaction exclusive</td>
<td>Transaction shared</td>
<td>Transaction shared</td>
<td>If not in transaction state</td>
</tr>
<tr>
<td>Transaction exclusive</td>
<td>Transaction shared</td>
<td>Transaction exclusive</td>
<td>If in transaction state</td>
</tr>
<tr>
<td>Transaction shared</td>
<td>Session shared</td>
<td>Session shared</td>
<td>Attempted</td>
</tr>
<tr>
<td>Transaction shared</td>
<td>Session exclusive</td>
<td>Session exclusive</td>
<td>Attempted</td>
</tr>
</tbody>
</table>

**Lock Contention Information**

JADE provides the LockContentionInfo class, which is a transient class used to record information about lock contentions (that is, when an attempt to lock a persistent object is queued or rejected because the object is already locked). When lock contention recording is started, lock contention information is recorded on the database server node.
Use the `System` class `beginLockContentionStats`, `clearLockContentionStats`, and `endLockContentionStats` methods to start and stop recording lock contentions and the `System` class `getLockContentionStats` and `getLockContentionInfo` methods to retrieve recorded lock contention information.

The properties defined in the `LockContentionInfo` class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>maxWaitTime</code></td>
<td>The longest time in milliseconds that any process spent queued waiting</td>
</tr>
<tr>
<td></td>
<td>to obtain a lock on the object</td>
</tr>
<tr>
<td><code>totalContentions</code></td>
<td>The number of lock contentions recorded for the object</td>
</tr>
<tr>
<td><code>totalWaitTime</code></td>
<td>The total time in milliseconds that all processes spent queued waiting</td>
</tr>
<tr>
<td></td>
<td>to obtain locks on the object</td>
</tr>
</tbody>
</table>

The method defined in the `LockContentionInfo` class is summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>target</code></td>
<td>Returns a reference to the object to which the lock contention information relates</td>
</tr>
</tbody>
</table>

**Unlocking Objects**

You can unlock objects manually. Use the `Object` class `unlock` method to unlock objects that are locked before a `beginTransaction`, `beginLoad`, or `beginTransientTransaction` instruction.

Objects that are requested to be unlocked after a `beginTransaction`, `beginLoad`, or `beginTransientTransaction` instruction are not unlocked until a `commitTransaction`, `endLoad`, `commitTransientTransaction`, or `abortTransaction` instruction. (For details about read and write transactions, see "Using Read and Write Transactions" under "Transaction and Concurrency Control Instructions", in Chapter 1.)

All persistent object updates must be preceded by a `beginTransaction`. As objects are updated, the JADE Object Manager maintains a list of modified objects. At `commitTransaction`, all modified objects are passed back to the server and all locks are released, starting the server transaction. If the application does an `abortTransaction`, the list of modified objects is discarded and all locks are released. (For details about sharing uncommitted persistent objects, see "Sharing Uncommitted Persistent Objects", in Chapter 1.)

**Notes** A `commitTransaction` instruction implies both an `endLock` and `endLoad` instruction.

In persistent transaction state, all unlock requests for persistent objects are ignored. Similarly, in transient transaction state, all unlock requests for shared transient objects are ignored. A session lock is therefore not released if the unlock request is made while in transaction state. To release a session lock, the unlock request must be made while not in transaction state.

Shared transient objects are automatically locked as they are updated within a transient transaction. The locks on these objects are released at the next `commitTransientTransaction` instruction.

Updates to shared transient objects are applied when the `commitTransientTransaction` instruction is executed. (For more details, see "Transaction and Concurrency Control Instructions", in Chapter 1.)

Use the `beginLock` and `endLock` instructions to bracket a read-only transaction, and ensure that objects referenced after executing the `beginLock` instruction and prior to the `endLock` instruction are the latest editions of the objects and remain unchanged during that interval. As objects are referenced, an implicit shared lock is acquired on each object, which causes the latest edition to be fetched from the server, if required.
Using Object Editions

Use the `Object` class `edition` and `latestEdition` methods to check whether the object in cache is the latest edition of that object and use the `resynch` method to ensure that you have the latest edition without requiring a lock. (When accessing objects without locking, a process can access an object even when another process is updating that object. The process does not see the updates, but instead views the most recently committed edition of the objects. This applies to persistent and shared transient objects.)

When you resynchronize an object (by using the `resynch` method), it simply marks the object in local cache as obsolete. The next time the object is referenced, JADE goes to the server node with the edition of the cached object. If it is not the latest edition, the updated object is brought back. If the cached object is already the latest edition (that is, the object has not been updated since it was brought into local cache), nothing is brought back.

As it is inefficient, do not use the `latestEdition` method to duplicate the edition check that JADE does automatically the next time the object is referenced after a `resynch` method. (As the `resynch` method only marks the locally cached object as obsolete; nothing is brought back from the server until the object is next referenced. Therefore, if you never reference the object after a resynchronization, JADE will not go to the server again for that object.)

**Tip**  If you know that you want to resynchronize an object, simply use the `resynch` method. Let JADE do the edition check for you the next time the object is referenced.

In general, it is better to have an object resynchronize another object rather than have the object resynchronize itself. For example, if you want to resynchronize a `myObj` object, `resynchObject(myObj)` is preferable to the `myObj.resynch` code.

If `myObj` is not in local cache (it may never have been referenced before or it was in cache but the cache has overflowed), using `myObj.resynch` will require bringing it into cache because it is the receiver of the `resynch` method. The object will then immediately be marked as obsolete by the resynchronization, requiring a trip to the server on the next reference (if only to confirm that it is the latest edition).

Using `resynchObject(myObj)` will do nothing if the `myObj` object is not in cache because there is no locally cached object to resynchronize. The next time `myObj` is referenced, it will be brought from the server anyway.

**Tip**  For the greatest efficiency resynchronizing other objects, use the current receiver of the method that you are in; that is, the `resynchObject (myObj)` example above.
Chapter 7  Using the Database Administration Framework

This chapter covers the following topics.

- Overview
- Using the Supplied Database Administration Framework
  - DbFile Class Constants, Properties, and Methods Summary
  - JadeDatabaseAdmin Class Constants and Methods Summary
  - Simple Scripted Database Backup
  - Developing Backup Applications
  - Initialization
  - Processing Backups Asynchronously
  - Notification and Progress Dialog Handling
  - Backup Transactions and Partial Database Backups
  - Exception Handling
  - Final Housekeeping Tasks
- Using the Online Database Backup Service
  - Creating and Using the Backup Database Dialog in User Application Logic
  - Using the Supplied Online Backup Dialog

Overview

The **JadeDatabaseAdmin** and **DbFile** system classes provide a database administration framework that encapsulates the behavior required to build standalone or integrated database administration tools for your JADE applications; for example, online database backups.

Online backups can be performed when the database is active for both read and write access. As recovery journal files are used during the recovery process to recover the database to a fully consistent state, recovery journals must be retained and the database informed when the backup starts and completes. All file backups must occur within the bounds of a backup transaction.

The **JadeDatabaseAdmin** class enables you to create backup tools that provide:

- Full online backup to disk. (Backup to other media is not supported.)
- Support for multiple backup destinations, backup concurrency, and data compression. (No support is provided for partial or incremental backup, or for third-party backup tools.)
- A simple backup service in the **RootSchema**.

**Tip**  Although you can use the administration framework provided by the **DbFile** and **JadeDatabaseAdmin** system classes to build standalone database administration applications, you can integrate the online backup service provided by JADE directly in any of your applications, to reduce your development work.
You can backup multiple files concurrently. Concurrent file backups enable your backup applications to take advantage of parallel I/O, which you can use to reduce the elapsed time required for a backup. Your backup applications are responsible for initiating simultaneous file backups, if required, using multiple JADE processes.

Each file backup operation executes as a database method on the server, utilizing a separate server thread. However, because of increased disk contention and disk head movement, concurrent backup operations run slower if source and destination files are not on separate disk devices.

**Note** A one-to-one correspondence between application threads and server threads does not exist. A file backup task is scheduled to run from the pool of available server threads, which means that in some cases backup application threads will be blocked waiting for an available server thread.

<table>
<thead>
<tr>
<th>For details about…</th>
<th>See…</th>
</tr>
</thead>
<tbody>
<tr>
<td>JadeDatabaseAdmin and DbFile classes</td>
<td>Chapter 1 of the JADE Encyclopaedia of Classes</td>
</tr>
<tr>
<td>Event notifications</td>
<td>&quot;DbFile Class Event Notifications&quot; and &quot;JadeDatabaseAdmin Class Event Notifications&quot;, in Chapter 1 of the JADE Encyclopaedia of Classes</td>
</tr>
<tr>
<td>JADE development environment online backups</td>
<td>&quot;Backing Up Your JADE Development Environment&quot;, in Chapter 3 of the JADE Database Administration Guide</td>
</tr>
<tr>
<td>Backing up and restoring the database</td>
<td>&quot;Administering Your JADE Database&quot;, in Chapter 3 of the JADE Database Administration Guide</td>
</tr>
</tbody>
</table>

**Using the Supplied Database Administration Framework**

You can use the constants, properties, and methods provided by the DbFile and JadeDatabaseAdmin system classes administration framework to integrate online backup services into your own applications or to build standalone database administration applications.

This framework enables you to develop a more application-specific online database administration utility than the database backup service provided in the RootSchema.

To operate 'online', your database administration utility must be a standard JADE application.

You could develop a wizard-style interface to allow users to customize and set up schedules for repetitive backup operations or other database administration tasks, if required.

**DbFile Class Constants, Properties, and Methods Summary**

The DbFile class provides the constants listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Description</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackupOperationEvent</td>
<td>File backup operation event</td>
<td>1001</td>
</tr>
<tr>
<td>BackupProgressEvent</td>
<td>File backup progress event</td>
<td>1000</td>
</tr>
<tr>
<td>Kind_Control</td>
<td>Control files (that is, _control and _reorg database files)</td>
<td>1</td>
</tr>
<tr>
<td>Kind_Environmental</td>
<td>Environmental files (_locks, _environ, and _stats database files)</td>
<td>2</td>
</tr>
<tr>
<td>Kind_System</td>
<td>System files (that is, _system, _sysxrf, _sysgui, _sysint, _sysdev, _systools, _jadeapp, _jadedef, and _sysdef database files)</td>
<td>4</td>
</tr>
</tbody>
</table>
The **DbFile** class properties summarized in the following table can be used to develop your database administration user applications.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>database</td>
<td>Contains the name of the database</td>
</tr>
<tr>
<td>excludeFromBackup</td>
<td>Specifies whether the file is excluded from database backup</td>
</tr>
<tr>
<td>kind</td>
<td>Contains the kind, or category, of database file</td>
</tr>
<tr>
<td>path</td>
<td>Contains the database file path</td>
</tr>
</tbody>
</table>

The **DbFile** class methods summarized in the following table can be used to develop your database administration user applications.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backupFile</td>
<td>Backs up a physical database file</td>
</tr>
<tr>
<td>certifyFile</td>
<td>Initiates the certification of a database file</td>
</tr>
<tr>
<td>changeAccessMode</td>
<td>Changes the access mode of a file to read-only or updateable</td>
</tr>
<tr>
<td>compactFile</td>
<td>Initiates the compaction of a database file</td>
</tr>
<tr>
<td>getFileLength</td>
<td>Returns the size of a physical database file</td>
</tr>
<tr>
<td>getFileStatus</td>
<td>Returns the status of a physical database file during the backup process</td>
</tr>
<tr>
<td>getFreeSpace</td>
<td>Evaluates the available free space in a database file</td>
</tr>
</tbody>
</table>

For details, see "DbFile Class", in Chapter 1 of the *JADE Encyclopaedia of Classes*. In addition, you can use the **Database** class **getName** method to return the name of a database file.
JadeDatabaseAdmin Class Constants and Methods Summary

The JadeDatabaseAdmin class constants summarized in the following table can be used to develop your database administration user applications.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BackupAbortedEvent</td>
<td>4000</td>
<td>Multiple file backup terminated by the user</td>
</tr>
<tr>
<td>BackupCancelledEvent</td>
<td>8000</td>
<td>Multiple file backup has been cancelled by the user</td>
</tr>
<tr>
<td>BackupCompleteEvent</td>
<td>3000</td>
<td>Multiple file backup has completed normally</td>
</tr>
<tr>
<td>BackupFailedEvent</td>
<td>9000</td>
<td>Multiple file backup has failed</td>
</tr>
<tr>
<td>FileBackupCompleteEvent</td>
<td>2000</td>
<td>File backup has finished</td>
</tr>
<tr>
<td>FileBackupStartEvent</td>
<td>1000</td>
<td>File backup has commenced</td>
</tr>
<tr>
<td>JournalTransferEvent</td>
<td>6000</td>
<td>Recovery journal file has been transferred</td>
</tr>
<tr>
<td>Mode_Archive</td>
<td>4</td>
<td>Database is in archive mode (journal files are archived for database recovery)</td>
</tr>
<tr>
<td>Mode_Default</td>
<td>9</td>
<td>Database is in default mode</td>
</tr>
<tr>
<td>Mode_Shared</td>
<td>0</td>
<td>Database is in shared mode</td>
</tr>
<tr>
<td>RpsStorageMode_Full</td>
<td>0</td>
<td>Persistent full database replica RPS data store mode</td>
</tr>
<tr>
<td>RpsStorageMode_MappedExtent</td>
<td>1</td>
<td>Persistent mapped extent RPS data store mode</td>
</tr>
<tr>
<td>RpsStorageMode_WorkingSet</td>
<td>2</td>
<td>Temporary working set RPS data store mode</td>
</tr>
<tr>
<td>Usage_NoAudit</td>
<td>2</td>
<td>Database cannot be updated; that is, it is in quiesced mode</td>
</tr>
<tr>
<td>Usage_ReadOnly</td>
<td>1</td>
<td>Database cannot be updated; that is, it is in quiesced mode</td>
</tr>
<tr>
<td>Usage_Update</td>
<td>0</td>
<td>Changes can be made to the database</td>
</tr>
</tbody>
</table>

The JadeDatabaseAdmin class methods summarized in the following table can be used to develop your database administration user applications.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abortBackup</td>
<td>Terminates an online backup transaction</td>
</tr>
<tr>
<td>backupAllDbFiles</td>
<td>Backs up all database files to a common directory</td>
</tr>
<tr>
<td>backupDbFiles</td>
<td>Backs up selected file kinds to a common directory</td>
</tr>
<tr>
<td>backupJournal</td>
<td>Copies the specified recovery journal file to backup</td>
</tr>
<tr>
<td>beginBackup</td>
<td>Starts an online backup transaction</td>
</tr>
<tr>
<td>changeDbAccessMode</td>
<td>Changes the access mode of the database</td>
</tr>
<tr>
<td>closeCurrentJournal</td>
<td>Closes the current recovery journal file and switches to a new one</td>
</tr>
<tr>
<td>commitBackup</td>
<td>Commits an online backup transaction</td>
</tr>
<tr>
<td>commitCoherentBackup</td>
<td>Commits an SDS Secondary backup transaction with coherent (commit-consistent) state</td>
</tr>
<tr>
<td>disableByteProgressEvents</td>
<td>Disables operation and progress event notifications for the number of bytes of a file backed up</td>
</tr>
</tbody>
</table>
Method | Description
--- | ---
disableProgressEvents | Disables progress event notifications for file backups
enableByteProgressEvents | Enables operation and progress event notifications for the number of bytes of a file backed up
enableProgressEvents | Enables progress event notifications for file backups
getAbortJournalNumber | Returns the number of the journal containing the Begin Transaction record of the oldest active transaction
getAllDbFiles | Populates an array with references to database files
getArchiveJournalDirectory | Returns the name of the archive directory for transaction journals
getCreationTimestamp | Returns the date and time the database was created
currentJournalDirectory | Returns the current recovery journal file directory
currentJournalName | Returns the current recovery journal file name
currentJournalNumber | Returns the current recovery journal file number
dbFiles | Populates an array with references to files of selected kinds
getLatestBackupTimestamp | Returns the date and time the database was last backed up without error
getLatestFullBackupTimestamp | Returns the date and time the entire database was last backed up without error
getOpenTimestamp | Returns the most recent date and time the database was opened
isArchival | Specifies whether archival recovery is enabled for the database
verifyJournal | Verifies the consistency of the specified recovery journal file

For details, see "JadeDatabaseAdmin Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

Simple Scripted Database Backup

The simplest form of backup consists of backing up all database files in a synchronous fashion to a single backup location with no progress indications.

The following JADE script demonstrates how to use the backupAllDbFiles method provided by the JadeDatabaseAdmin class to accomplish this task. The backupAllDbFiles method manages backup transactions (beginBackup and commitBackup), enumerating files, and uses an exception handler to handle file not found exceptions. You could schedule such a method to run daily at a nominated time from a background JADE process (for example, a server application) using a JADE timer. The backup in this example excludes system files, as these files are not updated during normal operations.

backupDatabase Method

```java
backupDatabase();
vars
dba : JadeDatabaseAdmin;
droppedFiles : DbFileArray;
file : DbFile;
includeSysFiles, verifyFiles, compressFiles, allowOverwrite : Boolean;
quiesce : Boolean;
backupDirectory : String;
```
Developing Backup Applications

The partial class specifications in this section show the relevant properties and methods used in the method examples later in this chapter. The BackupDatabaseDlg and BackupManager classes operate in collaboration, with the BackupDatabaseDlg class responsible for user-interface and monitoring concerns and the BackupManager class responsible for initiating backup operations using an instance of the JadeDatabaseAdmin class. The DbAdmin class is a subclass of the Application class.

These method examples demonstrate the use of backup progress notifications, and as they exclude most user interface concerns, they assume that the ‘option’ attributes have been edited and set by the implementation of the BackupDatabaseDlg class.

DbAdmin Class Partial Specification

```
DbAdmin
{
    jadeMethodDefinitions
        backupDatabaseProcessor(backupManager: BackupManager);
    }
```
BackupManager Class Partial Specification

BackupManager
{
    attributeDefinitions
    // implementation attributes
    backupCancelled: Boolean protected;
    compressFiles: Boolean protected;
    defaultDirectory: String protected;
    includeSysFiles: Boolean protected;
    overwriteFiles: Boolean protected;
    quiescedBackup: Boolean protected;
    verifyFiles: Boolean protected;
    referenceDefinitions
    // public interface properties
    dba: JadeDatabaseAdmin readonly;
    droppedFiles: DbFileArray implicitMemberInverse, readonly;
    jademethodDefinitions
    // public interface operations
    backupHandler();
    cancelBackup();
    finalise(backupAborted: Boolean);
    initialise(backupDir: String; compress: Boolean; verify: Boolean;
                includeSystemFiles: Boolean; overwrite: Boolean;
                quiesce: Boolean) updating;
}

BackupDatabaseDlg Class Partial Specification

BackupDatabaseDlg
{
    constantDefinitions
    BackupAbort: Integer = 9;
    BackupCancel: Integer = 2;
    BackupComplete: Integer = 1;
    BackupFailure: Integer = 3;
    BackupProgress: Integer = 7;
    FileBackupComplete: Integer = 5;
    FileBackupStart: Integer = 4;
    UserCancel: Integer = 6;
    attributeDefinitions
    // implementation attributes
    backupAborted: Boolean protected;
    backupCancelled: Boolean protected;
    backupInProgress: Boolean protected;
    eventsSubscribed: Boolean protected;
    referenceDefinitions
    // implementation references
    backupManager: BackupManager protected;
    currentFile: DbFile protected;
    // various GUI controls
    c_backupDirectory: TextBox;
    c_compressFiles: CheckBox;
    c_includeSysFiles: CheckBox;
    c_overwriteFiles: CheckBox;
}
Initialization

The constructor and initialize methods shown in the following examples perform initialization steps that are required by the subsequent example methods.

BackupDatabaseDlg::initialise Method

```java
initialise() updating;
begin
    // Create and initialize a backup manager object. This is created as a
    // sharedTransient so that we can safely share the object between two
    // JADE processes, the current user interface application, and the
    // background backup application.
    beginTransientTransaction;
    create backupManager sharedTransient;
    backupManager.initialise(c_backupDirectory.text,
        c_compressFiles.value,
        c_verifyFiles.value,
        c_includeSysFiles.value,
        c_overwriteFiles.value,
        c_quiescedBackup.value);
    commitTransientTransaction;
    // Initialize progress bar control
    progressBar.partsInJob := 100;
    // Subscribe to the various backup events we are interested in
    beginNotification(backupManager.dba,
        JadeDatabaseAdmin.FileBackupStartEvent,
        Response_Continuous, FileBackupStart);
    beginNotification(backupManager.dba,
        JadeDatabaseAdmin.FileBackupCompleteEvent,
        Response_Continuous, FileBackupComplete);
    beginNotification(backupManager.dba,
        JadeDatabaseAdmin.BackupCompleteEvent,
        Response_Continuous, BackupComplete);
```

BackupManager Constructor

create() updating;
begin
  // Create a JadeDatabaseAdmin instance. If self is a shared
  // transient, this means we are potentially being shared between
  // processes so ensure that our dba is also shareable
  if self.isSharedTransient then
    create dba sharedTransient;
  else
    create dba transient;
  endif;
end;

BackupManager::initialise Method

initialise(backupDir: String;
compress, verify, includeSystemFiles: Boolean;
overwrite, quiesce: Boolean) updating;
begin
  // Save backup parameters for calls to database backup methods
  defaultDirectory := backupDir;
  compressFiles := compress;
  verifyFiles := verify;
  includeSysFiles := includeSystemFiles;
  overwriteFiles := overwrite;
  quiescedBackup := quiesce;
  // Enable backup progress events to occur in increments of 4% or greater
  dba.enableProgressEvents(4);
end;

Processing Backups Asynchronously

You may often want to process database backups asynchronously, using an asynchronous JADE process.

You can use server applications or the Application class startApplication method to start a new process to perform a backup or you can have separate JADE processes to perform concurrent file backups.

You can schedule backups to occur automatically at a specific time or frequency, by using a JADE timer to start the process.

The DbBackup::doBackup method in the following example calls Application::startAppMethod method to start a new DbAdmin application called DatabaseBackup, starting in the DbAdmin::backupDatabaseProcessor method and passing a reference to a BackupManager instance as the single method parameter.
The **DatabaseBackup** application is specified with application type of **Non-GUI** (set from the JADE development environment Define Application dialog), which allows it to run as a background process with no forms. (This method would typically be initiated from a user interface event such as a button-click.)

**BackupDatabaseDlg::doBackup Method**

```cpp
doBackup() updating;
begin
  initialise;
  backupInProgress := true;
  // Fire up a DatabaseBackup application.
  // Application type = (Non-GUI)
  // Schema = DbAdmin
  // Starting method = backupDatabaseProcessor (defined in the
  // application class of the current schema)
  // Note that our backupManager is a shared transient object, as the
  // DatabaseBackup application runs as an asynchronous process and can
  // both reference and update an array of dropped files we later access
  // back in the current application thread when the backup operation
  // completes.
  // While the DatabaseBackup application is busy processing, the current
  // application monitors the progress of the backup by using the various
  // event notifications we subscribed to, and waits for a
  // BackupCompleteEvent or a BackupAbortedEvent that triggers finalize
  // processing.
  app.startAppMethod("DbAdmin", "DatabaseBackup",
                    "backupDatabaseProcessor", backupManager, false);
  backupInProgress := true;
end;
```

**DbAdmin::backupDatabaseProcessor Method**

The **DbAdmin** application subclass **backupDatabaseProcessor** method in the following example starts in a new JADE process and simply calls the **backupHandler** method on the **BackupManager** instance, using the reference passed by the initiator.

```cpp
backupDatabaseProcessor(backupManager: BackupManager);
begin
  backupManager.backupHandler;
end;
```

**BackupManager::backupHandler Method**

The **BackupManager::backupHandler** method shown in the following example is the main driver, which simply calls the **JadeDatabaseAdmin::backupAllDbFiles** method to do most of the actual work.

```cpp
backupHandler();
begin
  // Call the backupAllDbFiles method of the JadeDatabaseAdmin class to
  // backup all database files. Because we are passing a reference to our
  // droppedFiles array and this can be updated within the method, we must
  // bracket the call in a transient transaction.
  beginTransientTransaction;
  dba.backupAllDbFiles(defaultDirectory, includeSysFiles, verifyFiles,
                        compressFiles, overwriteFiles, quiescedBackup,
                        droppedFiles);
```
commitTransientTransaction;
end;

**Notification and Progress Dialog Handling**

The following methods execute on the initiating application process (that is, the process that invoked the doBackup method), whereas the backup itself continues asynchronously on the DatabaseBackup application process.

**BackupDatabaseDlg::userNotify Method**

The following example of a notification callback shows the handling of the JadeDatabaseAdmin class FileBackupStartEvent, BackupAbortedEvent, BackupCompleteEvent, and backup progress events.

```java
userNotify(eventType: Integer; theObject: Object; eventTag: Integer;
           info: Any) updating;
begin
  if eventTag = BackupComplete then
    finalise(false /*completed ok*/);
    return;
  endif;
  if eventTag = BackupAbort or eventTag = BackupCancel then
    finalise(true /*aborted*/);
    return;
  endif;
  if backupCancelled then
    // The user has cancelled the backup, avoid processing any of
    // the events handled below
    return;
  endif;
  if eventTag = FileBackupStart then
    currentFile := info.DbFile;
    if currentFile <> null then
      beginNotification(currentFile, DbFile.BackupProgressEvent,
                         Response_Continuous, BackupProgress);
      updateProgress("Processing file...", currentFile.name, 0);
    endif;
  elseif eventTag = FileBackupComplete then
    currentFile := null;
  elseif eventTag = BackupProgress then
    if currentFile <> null then
      updateProgress("Processing File...", currentFile.name,
                     info.Integer);
    endif;
  endif;
end;
```
**BackupDatabaseDlg::updateProgress Method**

The following method is called from the notification callback to update progress information, including the operation being performed and the target file. The percentage of the backup job is represented by the `ProgressBar` control.

```java
updateProgress(oper: String; fname: String; percentDone: Integer) protected;
begin
    // Move the bar on our progress bar control
    progressBar.partsDone := percentDone;
    if percentDone = 0 then
        // Update labels showing operation and file name currently being
        // processed
        operation.caption := oper;
        fileName.caption := fname;
        // Make sure the dialog repaints updated controls
        refreshNow;
    endif;
end;
```

**BackupDatabaseDlg::cancelBackup Method**

The following method may be called as a result of a Cancel button click from the progress dialog, to allow users to cancel a backup operation.

```java
cancelBackup() updating, protected;
vars
dba : JadeDatabaseAdmin;
begin
    if backupInProgress then
        backupInProgress := false;
        backupCancelled := true;
        if currentFile <> null then
            updateProgress("Cancel Pending ...", "Current file: " &
                        currentFile.name, 0);
            // Stop further progress notifications
            endNotification(currentFile, DbFile.BackupProgressEvent);
            currentFile := null;
        endif;
        // Create a new transient dba to avoid referencing the dba
        // owned by our backupManager, which may be locked in
        // transaction state by our asynchronous backup process
        create dba;
        dba.abortBackup;
    else
        // The backup never started, so assume user just wants to
        // cancel out of the form
        unloadForm;
    endif;
epilog
    delete dba;
end;
```
Backup Transactions and Partial Database Backups

The following example demonstrates the use of the `beginBackup`, `commitBackup`, or `abortBackup` method to bracket a backup transaction, and also shows a usage of the `getDbFiles` method to enumerate user data and environmental files and the `DbFile` class `backupFile` method to backup each file individually with specified parameter options.

BackupManager::backupDataFiles Method

This method does not backup system or user schema files and as such, it is doing a partial database backup, which is useful in a read-only schema environment where system and user schema files are not updated.

This method is designed so that it can be executed synchronously by a direct call or asynchronously from a wrapper similar to the BackupManager class `backupHandler` method, described under "Processing Backups Asynchronously", earlier in this chapter.

```cpp
backupDataFiles() updating, serverExecution;
vars
    dbFile  : DbFile;
    dbfiles : DbFileArray;
begin
    // Arm our backup exception handler
    on Exception do backupException(exception);
    // Begin a backup transaction. The quiesce parameter is false, meaning
    // that a 'hot' or online backup will be performed, allowing online
    // updating to continue during the backup processing.
    dba.beginBackup(defaultDirectory, false /*==> online backup*/);
    // Obtain an array of references to user data and environmental files
    create dbfiles transient;
    dba.DbFiles(DbFile.Kind_User_Data + DbFile.Kind_Environmental, dbfiles);
    foreach dbFile in dbfiles do
        // Since we have enumerated environmental files, we must exclude files
        // whose excludeFromBackup attribute is set; for example, _environ.dat
        if not dbFile.excludeFromBackup then
            dbFile.backupFile(null, // use default directory
                              true, // verify during backup
                              true, // request data compression
                              false); // disallow existing files overwrite
        endif;
    endforeach;
    // Commit the database backup transaction, this takes the database
    // out of backup state and finalizes the backup
    dba.commitBackup;
epilog
    if process.isInExceptionState then
        // A fatal exception occurred during the backup and the activated
        // exception handler aborted the current action
        dba.abortBackup;
    endif;
    delete dbfiles;
end;
```
Exception Handling

The following is an example of an exception handler armed by the `backupDataFiles` method. This handler filters `file not found` and `user aborted` errors, and passes all other exceptions to a higher-level exception handler. Most other exceptions that could occur during a backup will be fatal. Missing files may be valid if a database file defined in the schema has never been created or it has been deleted and no new objects created in the file.

**BackupManager::backupException Method**

```java
backupException(ex: Exception): Integer updating, protected;
begin
    if ex.errorCode = JErr_DbFileNotFoundException or
       ex.errorCode = JErr_DbUserAbort then
        return Ex_Resume_Next;
    endif;

    // pass all other exceptions to the next handler in the chain
    return Ex_Pass_Back;
end;
```

**Final Housekeeping Tasks**

The `finalise`, `unload`, `getDroppedFileText`, and `delete` methods shown in the following examples perform wrap-up processing, including ending notifications, deleting transients, and reporting on the disposition of the backup.

**BackupDatabaseDlg::finalise Method**

```java
finalise(aborted: Boolean) updating, protected;
vars
    msg : String;
    file : DbFile;
begin
    // Just before we finish, provide the user with some backup information.
    if aborted then
        msg := "Database Backup Aborted";
    else
        msg := "Database Backup Completed";
    endif;

    if backupManager <> null then
        // Attempt to explicitly share lock our backupManager instance.
        // In case the site runs with a very low default lock timeout,
        // we need to allow enough time for the backup worker process to
        // release its lock on the shared transient instance.
        if tryLock(backupManager, Share_Lock, Transaction_Duration, 10000) then
            msg := msg & backupManager.getDroppedFileText;
        endif;
    endif;

    app.msgBox(msg & Cr, 'Message', MsgBox_Exclamation_Mark_Icon + 65536);
    backupInProgress := false;
    backupAborted := aborted;
    // the unload method ends subscribed notifications, if required
    unloadForm;
end;
```
BackupDatabaseDlg::unload Method

Certain housekeeping tasks are performed in the unload method, in case the dialog is shut down prematurely.

```cpp
unload() updating;
begin
  if eventsSubscribed then
    endNotification(backupManager.dba,
                    JadeDatabaseAdmin.FileBackupStartEvent);
    endNotification(backupManager.dba,
                    JadeDatabaseAdmin.FileBackupCompleteEvent);
    endNotification(backupManager.dba,
                    JadeDatabaseAdmin.BackupCompleteEvent);
    endNotification(backupManager.dba,
                    JadeDatabaseAdmin.BackupAbortedEvent);
  endif;
  if backupManager <> null then
    beginTransientTransaction;
    backupManager.finalise;
    delete backupManager;
    commitTransientTransaction;
  endif;
end;
```

BackupManager::finalise Method

```cpp
finalise();
begin
  // disable backup progress events
  dba.disableProgressEvents;
end;
```

BackupManager::getDroppedFileText Method

```cpp
getDroppedFileText(): String;
vars
  msg   : String;
  dbFile : DbFile;
begin
  if droppedFiles.size > 0 then
    msg := 'The following files were not backed up : ' &CrLf;
    foreach dbFile in droppedFiles do
      msg := msg & dbFile.name & '.dat' &CrLf;
    endforeach;
  endif;
  return msg;
end;
```

BackupManager Destructor Method

```cpp
delete() updating;
begin
  delete dba;
end;
```
Using the Online Database Backup Service

The JADE RootSchema provides a database backup service that you can incorporate directly in any of your applications to reduce your development work. This provides you with a small subapplication that you can integrate into your own JADE applications as a simple backup service to initiate a full online backup to disk.

The backup service is controlled by the modal Backup Database dialog that provides the user interface and spawns an asynchronous JADE process to perform the actual backup operations. As it is modal, the user interface is synchronous, whereas the backup thread operates asynchronously in the background. The user interface also provides a progress dialog that monitors the progress of backup operations and allows the backup to be canceled.

This backup service is controlled by the JadeBackupDatabaseDialog subclass of the Form class, which uses a separate process to do the backup.

Although you can use the online backup service directly from your user applications, it does not make use of the full potential of the DbFile and JadeDatabaseAdmin class framework.

Creating and Using the Backup Database Dialog in User Application Logic

You can create and use the Backup Database dialog provided by the administration framework backup service from user application logic, as shown in the following menu click event method:

```pascal
backupdatabase_click(menuItem: MenuItem input) updating;
vars
    form : JadeBackupDatabaseDialog;
begin
    create form;
    form.showModal;
end;
```

The following restrictions apply when using the supplied framework in your own applications.

- A single backup destination only is supported
- Multiple concurrent file backups are not supported
- Partial backup (file selection) is not supported

Using the Supplied Online Backup Dialog

The Backup Database dialog, provided by the backup service contained in the database administration framework, enables runtime databases to be backed up while the database remains active for both read and write access. Alternatively, you can change the database mode so that the database is locked for write access.

To use the Backup Database dialog

1. In the Backup Directory text box, specify the path of the destination directory in which your database files are to be backed up, as shown in the following example of the backup directory on the c: drive of your database server workstation.

```plaintext
c:\jade\backup
```

You must specify a directory that is valid on the server. If it does not exist, it is created.
Alternatively, click the adjacent browse button (indicated by the … points of ellipsis symbol) to display the common Browse for Folder dialog that enables the selection of the backup directory in which the backed up database files will be located.

2. In the Backup Mode group box, select the Quiesced - Read Only option button if you want a quiesced read-only backup operation. When you select this option, the database is placed in a quiescent state by first allowing current active transactions to complete, flushing modified buffers from cache to the stable database. In this state, physical database files contain all committed updates to the database and the files are opened in read-only mode with shared read access, allowing external backup processes to safely copy database files. In a quiescent backup, updating transactions are not permitted and attempts to execute database transactions raise a database exception. When a backup is performed in a quiescent state, the physical database files are guaranteed to contain all database updates and a quiesced backup does not require backup recovery.

By default, the Online - Updating Allowed option is selected, indicating that updates are allowed during the backup process. When restoring a database backed up fully online (that is, the database was available for both read and write access at the time of the backup), the restoration process requires the recovery of backed up transaction recovery journals. A backup recovery is required after restoring files that were fully backed up online by using this default option.

3. Check the Compress Files check box if database files are to be compressed as they are backed up. By default, database files are not compressed.

4. Check the Overwrite Existing Files check box to allow any existing files in the specified destination backup directory to be overwritten as the database is backed up. By default, existing files are not overwritten.

5. Click the OK button. Alternatively, click the Cancel button to abandon your selections.

**Tips** A fast backup is performed if files are neither verified nor compressed. In a fast file backup, database files are backed up in a similar fashion to a standard file copy, using large buffers and asynchronous I/O to speed up the copy process. The fast backup mode bypasses the database access-routines and cache management, and does not verify data as it is backed up.

You must set the help file that you require (for example, the JADE.pdf file or your own help file) in the Help File text box in the Application sheet of the Define Application dialog.

If no errors are detected, the backup proceeds.

For details about using this same dialog to backup your JADE development environment database, see "Backing Up Your JADE Development Environment", in Chapter 3 of the JADE Database Administration Guide.
This chapter covers the following topics.

- **Overview**
- **Packages Usage Example**
  - Exported Package Example
  - Imported Package Example
  - Locally Defined Imported Class Method Example
- **Exported and Imported Packages**
  - Exported Package Overview
  - Imported Package Overview
  - Environmental Objects
  - Locales
  - Timer, Notification, and Control Event Methods
  - Extracting and Loading Package Schemas
- **Exporting a Package Using the Export Package Definition Wizard**
  - Using the Packages Menu for Exported Packages
  - Using the Export Package Definition Wizard to Define a Package
  - Changing a Package by Using the Export Package Definition Wizard
  - Browsing and Maintaining an Exported Package
  - Displaying References to an Exported Package
- **Importing a Package**
  - Using the Packages Menu for Imported Packages
  - Adding an Imported Package to Your Schema
  - Changing an Imported Package
  - Browsing and Maintaining an Imported Package
- **Stub Packages**
- **Extracting a Package**
- **Removing a Package**
- **Switching Application Contexts When Invoking a Method**
- **Calling User Methods from Packages**
Overview

Packages enable you to group reusable classes (implemented in one or more schemas) as an exported package and make them available to multiple unrelated schemas as an imported package.

Packages allow one schema (the importing schema) to import and make use of classes from another schema (the exporting schema). The importing and exporting schemas do not have to be related and they can exist in completely independent schema branches.

Note In this chapter, the term package refers to the mechanism by which reusable classes are exposed from an exporting schema or reused in an importing schema.

The exporting schema defines the classes that it exports in packages. A single schema can export multiple packages. You can include only classes with public access in a package. When you export a class, you can select the properties, methods, and constants that are made available to importing schemas.

The import name of a package, which is defined at the time it is imported, has the exported package name by default. An imported package does not have to have the same name as the exported package that it imports. However, you cannot change package definitions and export characteristics at import time. Imported classes implicitly inherit from the nearest RootSchema subschema copy class of the importing schema. You cannot subclass or re-export imported classes.

When an imported class is unambiguously identified (that is, its name does not conflict with a class already defined in the branch of the importing schema or with another imported class), you can access that class by name. When the name of an imported class conflicts with another class, access the class by prefixing the class name with the name of the imported package. When an imported class has the same name as a local class, notifications are registered for the local class.

For details about the :: operator that allows you to qualify imported class names, see "Accessing Meta Data", in Chapter 1.

Packages Usage Example

The Schema Browser in the following example contains application schemas extending down the DemoCompanySchema branch and a RootSchema subschema called LoggingPackageSchema, which provides log file services we would like to be able to use from our application schemas.
To use the log file services in `LoggingPackageSchema` in this hierarchy without using a package, we would have to take one of the following actions.

- Insert `LoggingPackageSchema` into the application schema hierarchy above the first schema in which we want to use the services. This can encourage top-heavy schema hierarchies.

- Make use of global instance visibility and indirect methods (for example, `getSchema`, `getClass`, `getPropertyValue`, `sendMsg`, and so on) to access `LoggingPackageSchema` entities across schemas. This is a commonly implemented approach but the use of meta or indirect methods is more complicated and generally not type-safe at compile time.

By using a package, the `LoggingPackageSchema` facilities can be seamlessly and type-safely used by any other schema. For the purposes of this example, imagine that `LoggingPackageSchema` contains the `LogManager`, `Log`, and `LogTagDict` classes shown in the following schema file.

```plaintext
LogManager completeDefinition
{
    referenceDefinitions
        allLogs: LogTagDict explicitInverse, subId = 1, number = 1;

    jadeMethodDefinitions
        create() updating, number = 1001;
        createLog(
            logTag: String;
            appTag: String;
            fileName: String): Log number = 1005;
        delete() updating, number = 1002;
        deleteLog(log: Log io) number = 1006;
        finalize() number = 1003;
        initialize() number = 1004;
    }

Log completeDefinition
{
    attributeDefinitions
        appTag: String[52] number = 2;
        fileName: String[258] number = 3;
        fred: Character protected, number = 5;
        logTag: String[52] number = 1;

    referenceDefinitions
        myLogManager: LogManager explicitEmbeddedInverse, number = 4;

    jadeMethodDefinitions
        close() number = 1001;
        create() updating, number = 1003;
        delete() updating, number = 1004;
        log(txt: String) number = 1005;
        open() number = 1002;
    }

LogTagDict completeDefinition
{
}
}
```

MemberKeyDefinitions

```plaintext
LogTagDict completeDefinition
{
}
```
DevRef - 2018.0.01

Chapter 8   Using Packages

```java
appTag;
logTag;
)

exportedPackageDefinitions
  Logging
  {
    exportedClassDefinitions
      Log
      {
        exportedPropertyDefinitions
          appTag;
          fileName;
          logTag;
        exportedMethodDefinitions
          close;
          log;
          open;
      }
    LogManager
      {
        exportedPropertyDefinitions
          allLogs;
        exportedMethodDefinitions
          createLog;
          deleteLog;
      }
  }

LogTagDict
{
}

referenceDefinitions
  myExceptionLog:   Logging::Log protected, number = 2;
  myLogManager:    Logging::LogManager protected, number = 1;
  myMsgLog:        Logging::Log protected, number = 4;
  myPollingLog:    Logging::Log protected, number = 3;

appInit
{
  appInit() updating;
  begin
    create self.myLogManager transient;
    self.myExceptionLog := self.myLogManager.createLog('DEMO',
      'ExLog', 'c:\temp\ex.log');
    self.myPollingLog := self.myLogManager.createLog('DEMO',
      'PollLog', 'c:\temp\poll.log');
    self.myMsgLog := self.myLogManager.createLog('DEMO',
      'MsgLog', 'c:\temp\msg.log');
    self.myExceptionLog.open;
    self.myPollingLog.open;
    self.myMsgLog.open;
    self.myMsgLog.log('Opened logs ' &
      self.myExceptionLog.fileName & ', ' &
      self.myPollingLog.fileName & ', ' & self.myMsgLog.fileName);
  end;
```
Although LoggingPackageSchema can also contain numerous implementation classes, we are not concerned with these in this packages usage example.

Exported Package Example

As the developer of LoggingPackageSchema, you decide that you want to make the public and read-only features (excluding create and delete methods, which cannot be exported) of LogManager, Log, and LogTagDict available for other schemas to use.

To do this using packages

1. Define a package called Logging. Schemas can export multiple packages, but we only need one in this example.

2. Add the LogManager class to the Logging package. When you add this class, you can specify the features of the class that you want to export so that you define an exported LogManager class, as follows.

```ruby
LogManager
{
    exportedPropertyDefinitions
        allLogs;
    exportedMethodDefinitions
        createLog;
        deleteLog;
}
```

Note that only a subset of the LogManager implementation has been exported (create, delete, initialize, and finalize methods have not been exported). The exported LogManager class effectively provides an interface to services provided by the implementation LogManager class.

3. Add the Log class to the Logging package and specify the features to be exported, as shown in the following pseudo schema file.

```ruby
Log
{
```
exportedPropertyDefinitions
  appTag;
  fileName;
  logTag;
exportedMethodDefinitions
  close;
  log;
  open;
)

Again, only a subset of the Log class has been exported.

4. Add the LogTagDict class to the Logging package, as follows.

   LogTagDict
   {
   }

5. Specify the application that will be used at run time to initialize the package when it is opened (that is, when a process that uses the package is initiated) and to finalize the package when it is closed (that is, when the process terminates). When the package is opened, the initialize method of the specified application is invoked and when the package is closed, the finalize method of the specified application is invoked. You can use these methods to perform package-specific initialization and finalization logic.

The Logging package is now ready to be imported for use in another schema.

Imported Package Example

As a developer in DemoApplicationSchema, you now want to make use of the services provided by the Logging package. Use the JADE development environment to locate the Logging package and then import it into DemoApplicationSchema. This imports the LogManager, Log, and LogTagDict classes into DemoApplicationSchema, with the features exported by the LoggingPackageSchema developer. You can now use these classes and features in DemoApplicationSchema type-safely, as if they were locally defined classes. For example, you could now define the following references in DemoApplicationSchema to your application class.

   myLogManager : LogManager;
   myExceptionLog : Log;
   myPollingLog : Log;
   myMsgLog : Log;

Class names imported in a package may conflict with locally defined classes. In such cases, you can use the package name to resolve name conflicts. For example, if LogManager and Log conflicted with locally defined classes, you could define the above references as follows.

   myLogManager : Logging::LogManager;
   myExceptionLog : Logging::Log;
   myExceptionTest : Testing::Test;
   myPollingLog : Logging::Log;
   myMsgLog : Logging::Log;

Having defined the above references, you could write finalize and initialize methods like those shown in the following examples.

   appInit() updating;
   begin
     create self.myLogManager transient;
     self.myExceptionLog := self.myLogManager.createLog('DEMO',

'ExLog', 'c:\temp\ex.log');
self.myPollingLog := self.myLogManager.createLog('DEMO',
    'PollLog', 'c:\temp\poll.log');
self.myMsgLog := self.myLogManager.createLog('DEMO',
    'MsgLog', 'c:\temp\msg.log');
self.myExceptionLog.open;
self.myPollingLog.open;
self.myMsgLog.open;
self.myMsgLog.open;
self.myExceptionLog.open;

Notes  The Process class finalizePackages and initializePackages event methods enable you to perform initialization and termination functions in package applications. If you run a JadeScript or a Workspace method and you do not call these events, initialize and finalize events for the application are not executed.

JADE does not initialize any packages for RootSchema applications running in user schemas, including unit tests run using the JADE Unit Test framework and the default RPS Datapump application in the RootSchema. If you require this initialization, you must initialize packages in your application code (using the Process class initializePackages and finalizePackages methods).

Locally Defined Imported Class Method Example

You can now add your own methods to imported classes in DemoApplicationSchema, in much the same way that you add methods to subschema copies of superschema classes. For example, note that in the appFini method in the previous "Imported Package Example" subsection, deleteLog is invoked for every Log object.

In DemoApplicationSchema, you can write your own method on LogManager to delete all log files, as shown in the following example.

```
LogManager::deleteAllLogs() updating;
vars
    log : Log;
begin
    foreach log in self.allLogs do
        self.deleteLog.log;
    endforeach;
end;
```

Note that deleteAllLogs is an extension of the imported class in the importing schema. With this locally defined method, you could write the appFini method as follows.

```
appFini() updating;
begin
    self.myLogManager.deleteAllLogs;
    delete self.myLogManager;
end;
```
Chapter 8  Using Packages

Exported and Imported Packages

A schema can export multiple packages, with each package having a name that is unique within its exporting schema (across both exported and imported packages). A package name does not have to be globally unique.

For clarity of implementation and use, subschemas of exporting schemas cannot inherit packages exported from superschemas. Any functionality exported by a schema is already available to subschemas via schema inheritance.

For an exported package to be used in another schema, that schema must import it.

For an overview of restrictions when exporting or importing packages, see "Summary of Restrictions when Using Packages", later in this chapter.

Exported Package Overview

You can specify the classes exported in a package. Classes imported by a schema cannot be exported from that schema. You can selectively export properties, methods, and constants of a class. When a class is exported, some class and feature attributes may be overridden. For example, you can change the allowed lifetime and default persistence of a class, or change the access of a public property to read-only.

You can specify the application that is to be instantiated when a package is opened so that package-specific initialization and finalization logic is performed.

The classes exported in a package effectively define a set of interfaces to the package schema. The exported classes should be self-contained in the branch of the schema that defines the package (although the implementation of those classes may rely on other inherited or imported classes).

Packages can export public classes inherited from superschemas so that you can divide the implementation of a package or packages across multiple schemas and provide a lowest-level subschema from which all packages are exported. Importers of the package or packages need be aware of only one schema (the lowest-level schema that exports all packages), rather than having to imported packages from several implementation schemas.

As you can export classes inherited from superschemas, you must consider subschema copy classes. Consider the following schema and class hierarchy.

```
SchemaA
├── Object
│    └── ClassA
|       └── ClassB

ClassA provides method m1.
ClassB provides method m2.
```

```
SchemaB (subschema of SchemaA)
├── Object
│    └── ClassA
|       └── ClassB

ClassB provides method m3.
```

In this example, `SchemaA::ClassA` and `SchemaA::ClassB` are root classes, as they are the top-level definition of those classes. `SchemaB::ClassA` and `SchemaB::ClassB` are subschema copy classes, as they are definitions below the top-level definitions. Method `m3` is a subschema copy method, as it is a method defined on a subschema copy class.
Chapter 8  Using Packages

If `ClassB` is to be exported in a package, the methods that are available for export depend on whether it is the root class being exported or the subschema copy class. If `SchemaA::ClassB` is exported, method `m2` is available for export. If `SchemaB::ClassB` is exported, methods `m2` and `m3` are available. As class names must be unique within a package, a single package cannot export both a root class and subschema copies of that root class. If `SchemaA::ClassB` were exported in a package and it later became necessary to export method `m3`, `SchemaA::ClassB` would need to be removed from the package and `SchemaB::ClassB` added.

**Note** In the above example, method `m1` cannot be exported on `ClassB`. To export method `m1`, `ClassA` must be exported or method `m1` must be reimplemented on `ClassB` (possibly performing only an `inheritMethod` call).

These issues do not exist when exporting properties, as properties can be defined only on root classes.

You cannot export a `RootSchema` subschema copy class. Exporting subclasses of `Application`, `Global`, and `WebSession` is not permitted.

For details about exporting a package, see "Exporting a Package Using the Export Package Definition Wizard" or "Browsing and Maintaining an Exported Package", later in this chapter.

**Imported Package Overview**

A schema can import multiple packages. Each imported package must have an import name that is unique within its importing schema branch (across both imported and exported packages) but the imported package name does not have to be globally unique.

Although an imported package name is the same as the exported package name by default, you can change the name when you import the package.

An import name is effectively an alias for a schema-package path. For example, a schema might import a package with a name of `Logging` that maps to a `LoggingUtilities` package exported from a `RootSchema::BaseSchema::LoggingSchema` schema. The import name provides a layer of indirection between the consumer of a package (the importer) and its producer (the exporter). It allows package name conflicts to be resolved by the importer of the package (where an importing schema wants to import two packages with the same name from different producers).

When referring to imported classes, code in the importing schema can prefix imported class names with the imported package name rather than having to specify the full schema path to the package. For details about the :: scope operator that allows you to do this, see "Accessing Meta Data", in Chapter 1.

A schema that exports a package cannot also import it. Packages imported by a schema are available to subschemas of the importing schema. Classes imported by a schema cannot be exported from that schema. You cannot change imported constants.

While local (importing schema) classes can define references to imported classes, imported classes cannot participate in inverse relationships with locally defined classes. Doing so would require modifying the structure of the imported class (to add inverse references), which is not permitted. It would also introduce a very tight (and potentially restrictive) coupling between an exporting schema and its importers.

For details about importing a package, see "Importing a Package", later in this chapter. See also the following subsections.

- Extending Imported Classes
- Implicit Inheritance from Nearest RootSchema Subschema Copy Class
- Preventing Multiple Imports of the Same Class
- Persistent Imported Class Instances
Extending Imported Classes

JADE does not permit you to subclass imported classes.

**Tip** Where two class hierarchies require a tightly coupled, modular, subclassed relationship, use schema inheritance (allowing subclass relationships) instead of imported packages and take advantage of global instance visibility. Subschema copy methods, schema inheritance, and global instance visibility may be more appropriate for schema entity reuse than imported packages in some instances.

If you are a developer of importing schemas and you want to extend the state of imported classes, you can do so by using aggregation. This might involve creating a local class that references an imported class and defines additional state or if persistent instances of an imported class are not permitted, implement a local class that copies and makes imported class instances persistent in the importing schema.

This enables developers of exported packages control over the persistence of their classes, as follows.

- If persistent instances are not permitted, they know that any changes they make in future versions of their packages will affect (reorganize) only exporting schema classes. Importing schema local class instances will not require reorganization.
- If persistent instances are permitted, they know that any changes they make in future versions of their packages will affect (reorganize) only exporting schema classes and possibly local importing schema collections of those classes.

Other local importing schema classes will not be affected because subclasses are not allowed.

You can add methods and constants to imported classes in the importing schema.

Implicit Inheritance from Nearest RootSchema Subschema Copy Class

By default, imported classes are positioned under their nearest imported superclass (that is, their nearest superclass that is also exported in the package being imported).

When an imported class has no imported superclasses, it inherits implicitly from the importing schema’s nearest local RootSchema subschema copy class. For example, assume we have the following export hierarchy.

In this example, the classes displayed in green are exported as package P1.
If package `P1` is imported, the Class hierarchy of the importing schema is as follows.

```
Object
  ↘ Application
    ↘ C1
        ↘ C2
    ↘ C6
    ↘ Collection
      ↘ Etree
        ↘ Dictionary
            ↘ MemberKeyDictionary
                ↘ Dict1
    ↘ Global
        ↘ JadeHTMLClass
            ↘ Html1
    ↘ HttpSession
```

`Dict1` is positioned under the `MemberKeyDictionary` class, which is its nearest `RootSchema` subschema copy superclass. `Html1` is positioned under the `JadeHTMLClass` class, which is its nearest `RootSchema` subschema copy superclass.

`C4` is positioned under `C2`, as `C3` has not been imported. (`C3` is not exported in package `P1.`) `C6` is positioned directly under the `Object` class, as its superclass (`C5`) has not been imported. The `Object` class is the nearest `RootSchema` subschema copy superclass of `C6`.

Imported classes inherit implicitly from the nearest `RootSchema` subschema copy classes of the local importing schema, and local subschema copy methods are available to them. You can therefore write methods that operate on local class instances and imported class instances in a consistent manner.

**Note** There is no direct inheritance as there is for local schema classes.

Although local (importing schema) `Object` methods can have the same name as imported methods, such methods are not polymorphic and they can have different signatures.

**Method Dispatch**

JADE recognizes when a subschema copy method is being invoked and binds the method in a different schema branch, if required. See also "Calling User Methods from Packages", later in this chapter.

As imported classes inherit implicitly from the importing schema’s nearest local `RootSchema` subschema copy class and local methods can be added to imported classes, the following class hierarchies are permitted.

**ExportSchema**

```
ExportSchema
 ↗ Object
      ↘ ClassA
          ↘ ClassB
```

The `Object` class provides method `m1`. `ClassA` reimplements method `m1` (exported).

**ImportSchema**

```
ImportSchema
  ↘ Object
      ↘ ClassA
          ↘ ClassB
```

`ClassB` provides method `m2` (exported).
The **Object** class provides local method **m1**.
**ClassA** provides method **m1** (imported).
**ClassB** provides method **m2** (imported) and local method **m3**.

Given a process running out of **ImportSchema** and an instance of **ClassB**, we have several use cases, as follows.

- Invoking method **m2** always executes **ExportSchema::ClassB::m2**.
- If **ExportSchema::ClassB::m2** invokes method **m1**, it always executes **ExportSchema::ClassA::m1** so that an importing schema cannot circumvent the implementation of the package.
- Invoking method **m3** always executes **ImportSchema::ClassB::m3**.
- If **ImportSchema::ClassB::m3** invokes method **m1** without qualification, it always executes **ImportSchema::Object::m1**. For example, in method **m3**, **self::m1** always executes **ImportSchema::Object::m1**. The compiler resolves method calls against locally defined methods first. In this example, **ImportSchema::Object::m1** is the nearest locally defined method so it is this method that is called.
- If **ImportSchema::ClassB::m3** needs to invoke the imported **ImportSchema::ClassA::m1**, it does so by using a new **importMethod** reserved word (keyword), for example:

  ```
  result := importMethod m1();
  ```

In cases where a locally defined method has the same name as an exported method, developers in the importing schema can use the **importMethod** reserved word to specify that it is the imported method that is to be invoked.

- Consider a method on a local (that is, not imported) **ImportSchema** class that receives an object via a parameter and must invoke the local **ImportSchema::Object::m1** method on the supplied object. If this method is to receive **ClassB** instances, the parameter must be of type **ClassB** or **Object**. In either case, because of the rule that locally defined methods are resolved first, the local **Object::m1** method is always invoked. This rule also applies to object references and local variables.

From these use cases, the following rules apply.

- Given a reference to an instance of an imported class and an imported method on that class has the same name as a locally defined superclass method, you must use the **importMethod** keyword to invoke the imported method via that reference.
- If **ref** holds a reference to an imported class instance at run time, **ref::m1** always invokes the local **Object::m1** method because locally defined methods are resolved first. Consider the following conditions.
  - A reference, **ref**, of type **Object**
  - An imported class, **C**
  - A local subschema copy method, **m1**, defined on **Object**
  - An imported method, **m1**, defined on **C**

Under these conditions, to invoke the **C::m1** via the **ref** reference, **ref** must be type-guarded to **C** and the **importMethod** keyword used as follows.

  ```
  ref.C importMethod m1;
  ```

The type guard is necessary because the **importMethod** keyword is permitted only when the type of the receiver, **ref**, is an imported class.
When invoking an Object subschema copy method via a reference of type Object and that method has the same name as an imported method, binding to the local Object method by default is necessary to preserve the behavior of existing code.

Consider the ImportSchema class hierarchy shown earlier in this section, with a local Object::m1 method and an imported ClassA::m1 method.

Imagine that we have the following ImportSchema application method, which existed before anything was imported:

```plaintext
doStuffToObjects(objSet: ObjectSet);
vars
  obj : Object;
begin
  foreach obj in objSet do
    obj.m1;
  endforeach;
end;
```

This method was written to invoke Object::m1 for each member of objSet, with the appropriate reimplementations being invoked if present. With packages, objSet may contain instances of imported classes, which may have their own m1 methods that are not reimplementations of the local Object::m1 method and may even have different signatures.

Binding first to locally defined methods ensures that such code does not need to be recompiled when classes are imported, and by default, continues to behave as intended when first written.

Expanding on the previous example, consider the case where the local Object class has implemented a method m1 and the imported class has both an imported method m1 and a local method m1, as follows:

```
ExportSchema
  Object
  ClassA

ClassA provides method m1 (exported)

ImportSchema
  Object
  ClassA

The Object class provides local method m1.
ClassA provides method m1 (imported) and reimplements local method m1.
```

This is permitted so that you have the full power of generic code like that shown in the following example.

```plaintext
foreach obj in objSet do  // where obj is of type Object
  obj.m1;
endforeach;
```

In this example, it is quite likely that the local Object method is abstract or that reimplementations are expected on a number of subclasses, including imported classes.

You can reimplement local superclass methods on imported classes to take advantage of polymorphism.
Example of Method Dispatch Using the importMethod Instruction

In the exporting schema where the class Html1 was created, methods like the updateValues method in the JadeHTMLClass class are reimplemented in the Html1 class.

When the updateValues method is imported, it is not regarded as the reimplementation of the updateValues method in the RootSchema subschema copy class JadeHTMLClass in the importing schema. You must provide your own reimplementation of updateValues in the imported Html1 class to call the imported implementation, as follows.

```javascript
updateValues(): Boolean updating;
begin
    return importMethod updateValues();
end;
```

Preventing Multiple Imports of the Same Class

A class can be imported only once into a specific schema branch from only one package. Consider the following exporting schema and exported packages. (Note that ClassB is exported in two packages.)

ExportSchema

```
Object
   - ClassA
   - ClassB
```

ClassA provides method m1 (exported)
ClassB provides method m2 (exported)

PackageA

```
ClassA
   - m1
   - m2
```

PackageB

```
ClassB
   - m2
```

This is valid from the point of view of the exporting schema as it is reasonable for an exporter to provide several packages or multiple versions of the same package, with common classes in each. It is the responsibility of the package importer to prevent multiple imports of the same class.

Now consider the following invalid importing schema and imported packages. ClassB has been imported twice in two different packages, and a local m3 method has been added twice, once on each imported class.

ImportSchema

```
Object
   - ClassA
   - ClassB
```

ClassA provides method m1 (imported).
ClassB provides method m2 (imported) and adds local method m3.
PackageB

ClassB provides method m2 (imported) and adds local method m3.

Method m3 would be ambiguous if multiple imports of ClassB were permitted.

Whenever a class is imported or a class is added to an exported package that has been imported, both the JADE development environment and compiler check that no other imported packages in the schema branch have that class. An exception is raised if there is an existing class with the same name.

It is the responsibility of the importer to prevent multiple imports of the same class. Class usage conflicts are marked as being in error when you import the package. For details, see "Importing a Package", later in this chapter.

**Persistent Imported Class Instances**

If an imported class allows persistent instances to be created, any persistent instances created by code in importing schemas reside in the map file defined on the exported class in the exporting schema.

When a class is imported by multiple schemas and applications in those schemas all create persistent instances of the imported class, a single map file (that of the exported class) contain instances from several applications.

**Environmental Objects**

When code in a package is compiled, it may need to access features of its local app, appContext, global, currentSession, and currentSchema system variables. Several sets of environmental objects or references are maintained for each process. There is one set for the schema of the process itself and one for each schema from which a package is imported.

The current schema context is maintained for each process. When a method is dispatched that is bound in a schema that differs from the current context, the current context is updated. Similarly, when a method returns, the current context is updated.

**Application** properties (app) that affect the behavior of a process are reflected across all application objects associated with that process, as they hold state that is relevant to the entire process and all packages running within it (for example, the printer, startupForm, aboutForm, showBubbleHelp, and userSecurityLevel properties, and so on).

If a process is using two packages from different schemas, for example, that process has three app objects (one for each imported package schema and one for the main process). As all application objects should have the same printer, for example, if the printer is changed by any code executed by the process, the change is reflected across all application objects.

**Note** If you have two packages (for example, p1 and p2), the second package (p2) imports the first package (p1), and a third schema imports both the p1 and p2 packages, there will be two instance of app for the first package (p1); that is, one in the context of the importing schema and the other in the second p2 package.

Similarly, WebSession objects (currentSession) hold state that is relevant to the entire process and all packages running within it (for example, the sessionId and lastAccessTime properties, and so on). If a process is using two packages from different schemas, for example, that process has three currentSession objects. However, as all session objects should have the same lastAccessTime property value, for example, if the last access time is changed by any code executed by the process, the change is reflected across all Web session objects.

ApplicationContext objects (appContext) hold state that is relevant to the entire process and all packages running within it. If a process is using two packages, for example, that process has three appContext objects.
Locales

Each Application object has a currentLocale reference. This reference refers to the Locale object that will be used to access forms and translatable strings from the application schema.

When a process is making use of one or more packages, the currentLocale references on the package Application objects may reference different Locale objects to the currentLocale of the main Application object for the process.

If you need to ensure that you always reference the currentLocale of the Application object of the main process, call process.getProcessApp().currentLocale.

Note The app.currentLocale reference is valid for applications that do not use packages.

If the schema of a form or translatable string does not contain the locale of the current process, the default locale is used.

Timer, Notification, and Control Event Methods

You can neither export nor re-implement event methods (for example, the timerEvent, userNotification, sysNotification, click, load, and unload event methods) in the importing schema.

Extracting and Loading Package Schemas

You can extract an exported package (as .scm and .ddb or .ddx files) and load it into another JADE database. The package can then be imported into another schema as soon as it has been loaded into the database; that is, a package is not visible for selection in the Select Package to Import list box on the Import Package dialog unless one of the following applies.

- It was exported from a schema in the same database.
- An exported package defined and extracted from another database has first been loaded into the current database.

Caution If you attempt to load an extracted schema containing imported classes and features before the schema containing the export schema is loaded, the schema load reports an error. You must therefore take care when loading a schema containing an imported package that you first load the schema containing the exported package. However, when loading a multiple extract file that contains both a schema containing an export package and a schema that imports that package, the dependency is understood at the time the multiple extract is performed and the schema containing the exported package is loaded first.

You can avoid problems of this nature if you maintain the exported package as a separate schema file from the schema or schemas into which it is to be imported. A user of the imported schema should first load the schema into which the package is to be imported and then load the separate package schema file into that schema. (For details about extracting an exported package, see "Extracting a Package", later in this chapter.)
The extraction of patches handles the extraction of partial packages and the loading of schemas handles partial packages. The following rules apply to the extraction of partial packages.

- A full extract of a package for a patch is performed when:
  - The package was created in that patch.
  - An exported class or interface was removed from the package in that patch.
  - Package options were updated (for example, changing the application).

  In all other cases, a partial extract is performed.

- A full extract of an exported class or interface for a patch is performed when:
  - The class or interface was added to the package in that patch.
  - The exported class options were changed in that patch.
  - A property, constant, or method belonging to the class was deleted in that patch.

  In all other cases, a partial extract is performed.

- There are no JADE command file (jcf) commands to handle partial packages.

To handle the extraction and loading of partial packages, patch control functionality now records changes to individual entities in the package. The patch control hook is therefore called for every entity that is added to the patch history when dealing with packages and patch control extensions are enabled.

**Exporting a Package Using the Export Package Definition Wizard**

The Export Packages Browser enables you to define and maintain an exported package.

**To open an Export Packages Browser**

1. Select the **Packages** command from the Browse menu.
2. Select the **Export Package** command from the submenu that is then displayed.

The Export Packages Browser window is then opened.

If you have not yet created an exported package for the current schema, nothing is displayed in the Export Packages Browser. Only one Export Packages Browser for the current schema can be open at any time. However, you can have concurrent open Export Packages Browsers for different schemas in a development work session.

You can change your default browser options, if required, by using the **Browser** sheet, accessed from the Options menu **Preferences** command.

For details about exported packages, see "Exported Package Overview" under "Exported and Imported Packages", earlier in this chapter. For details about using the Export Package Browser to define or maintain an exported package, see "Browsing and Maintaining an Exported Package", later in this chapter.
## Using the Packages Menu for Exported Packages

The Export Packages Browser provides the Packages menu, containing the commands listed in the following table, to enable you to define and maintain your exported packages.

<table>
<thead>
<tr>
<th>Command</th>
<th>For details, see…</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Using the Export Package Definition Wizard to Define a Package</td>
<td>Displays the Export Package Definition Wizard</td>
</tr>
<tr>
<td>Change</td>
<td>Changing a Package by Using the Export Package Definition Wizard</td>
<td>Displays the Export Package Definition Wizard</td>
</tr>
<tr>
<td>Remove</td>
<td>Removing a Package</td>
<td>Deletes the selected package</td>
</tr>
<tr>
<td>Browse</td>
<td>Browsing and Maintaining an Exported Package</td>
<td>Displays the Export Package Browser</td>
</tr>
<tr>
<td>Extract</td>
<td>Extracting a Package</td>
<td>Extracts the selected exported package</td>
</tr>
<tr>
<td>References</td>
<td>Displaying References to an Exported Package</td>
<td>Displays the schema that import the package</td>
</tr>
</tbody>
</table>

The actions that you can perform by using Packages menu commands for exported packages are described in the following sections.

### Using the Export Package Definition Wizard to Define a Package

The Export Packages Browser enables you to add one or more user-defined classes in the current schema to an exported package.

You cannot export any of the following.

- A system (that is, `RootSchema`) class; for example, the `Printer` or the `Form` class
- `RootSchema` subschema copy classes
- Subclasses of the `Application`, `Global`, or `WebSession` class
- A class whose access type is `Protected`
- Constructor (`create`) or destructor (`delete`) methods

#### To add an exported package to the current schema

1. If the Export Packages Browser does not have focus, select the **Export Package** command from the Packages submenu in the Browse menu.
2. Select the **Add** command from the Packages menu in the menu bar or the popup (context) menu when you right-click in the Export Packages Browser. Alternatively, select the **Add** command from the Packages menu.

The Export Package Definition Wizard is then displayed.

**Note** The Export Package Definition Wizard is a wizard-style dialog that consists of five steps, each represented by a sheet of the dialog.

Use the **Next >** and **< Back** buttons to navigate forwards or backwards through the steps. No step is enabled until the previous step has been completed.
The Export Package Definition Wizard sheets are described in the following subsections.

- Defining the Package
- Creating a Working Set of Classes
- Selecting Classes and Features
- Selecting Interfaces
- Changing Attributes

**Defining the Package**

The **Define Package** sheet of the Export Package Definition Wizard, shown in the following example, is displayed when you first access the wizard.

To define your exported package

1. In the **Package Name** text box, specify a name for the exported package that is unique within the current schema (that is, it cannot be the same as that of another exported package or imported package in the schema).

2. In the Select Schema group box, select the **Superschemas Up To** option button if you do not want your exported package made up only of classes or features in the current schema. In the drop-down list box, select the highest superschema in the hierarchy whose classes and features are also to be available for selection. You are also selecting the highest superschema in the hierarchy whose application is available for section in the following step.
3. In the **Select Application** drop-down list box, select the application in the schema that is to be instantiated when the package is opened at run time. This enables you to perform specific initialization and finalization logic when a package is initiated and terminated, respectively.

   When the application is instantiated, its initialization method is invoked and when it is terminated (when the process using the package terminates), its finalization method is invoked.

   If you require specific initialization or finalization logic to be performed, you can define a new application with the required methods and specify this as the exported package application.

4. In the **Package Description** text box, enter descriptive free-format text for your exported package, if required. This may be useful to help a package importer determine the package that he or she wants to import if there are several packages that have similar names.

5. Click the **Next >** button when you have defined the package you want to create.

   Alternatively, click the **Cancel** button to abandon your selections and close the wizard.

### Creating a Working Set of Classes

When you have defined the package that you want to create and clicked the **Next >** button, the **Create a Working Set of Classes** sheet of the Export Package Definition Wizard is then displayed, as shown in the following image.

The **Available Classes** list box contains all of the user-defined classes in the current schema (and superschemas, if applicable), with the exception of **RootSchema** subschema copy classes, classes whose access is not **Public**, and subclasses of the **Application**, **Global**, or **WebSession** class.

Classes that are not available for selection are disabled (for example, classes imported into the schema as these cannot be exported). From this list, select the classes that are to make up the working set of the exported package.
The **Working Set of Classes** list box contains the classes that you have selected for inclusion in your exported package working set. Classes that you select for inclusion in the working set are displayed in green in the **Available Classes** list box.

You must explicitly select each class for inclusion in the exported package. As no indirect exportation is assumed, exporting a class does not automatically export its superclass or subclasses.

**To locate a class**
- In the **Find Class** combo box, specify or select the class that you want to locate in the **Available Classes** list box.

That class is then displayed in the **Available Classes** list box, with the class hierarchy expanded, if applicable.

**To select the classes for your package working set, perform one of the following actions**
- Double-click a class in the **Available Classes** list box. Only the selected class is moved to the **Working Set of Classes** list box. If the selected class has subclasses, you must repeat this action for each subclass that you want to include in the working set.
- Select one or more of the classes listed in the **Available Classes** list box (by using the Shift or the Ctrl key to make multiple selections), and then click the single right-arrow (>) button. Only the selected class or classes are moved to the **Working Set of Classes** list box. If the selected class or classes have subclasses, you must repeat this action for each subclass that you want to include in the working set.
- To select a class and all of its subclasses, select the superclass (or superclasses) in the **Available Classes** list box and then click the double right-arrow (>>) button.

The selected classes are then automatically copied to the **Working Set of Classes** list box and are displayed in green in the **Available Classes** list box. You can select a class once only. An attempt to select a class that is already included in the working set has no effect.

Classes are added to the **Working Set of Classes** list box in alphabetical order and they do not have a class hierarchy once they are selected for inclusion in the working set. Your selections are not finalized until you click the **Next >** button. Before you click the **Next >** button, you can reverse any of your selections by removing them from the **Working Set of Classes** list box.

**To remove classes from the working set, perform one of the following actions**
- Double-click a class in the **Working Set of Classes** list box.
- Select one or more of the classes in the **Working Set of Classes** list box (by using the Shift or the Ctrl key to make multiple selections) and then click the single left-arrow (<) button.
- To select all classes in the **Working Set of Classes** list box, click the double left-arrow (<<) button.

The selected classes are then automatically moved from the **Working Set of Classes** list box to the **Available Classes** list box, where they are restored to the original class hierarchy and are displayed in black.

Click the **Next >** button when you have selected all classes for inclusion in the package working set. Alternatively, click the **< Back** button to redisplay the **Define Package** sheet or the **Cancel** button to abandon your selections and close the wizard.
Selecting Classes and Features

When you have selected the classes for inclusion in the exported package working set and clicked the Next button, the Select Classes and Features for Package sheet of the Export Package Definition Wizard is then displayed, as shown in the following image.

Use this sheet to select the public or read-only properties, methods, and constants associated with working set classes that you want to include in the exported package. (You cannot export protected properties and methods.)

Classes included in the working set are shown alphabetically in the Available Classes and Features list box. You can expand all class nodes to show all of their features or collapse them to show only the class name itself. An expanded node is displayed with a minus sign (-) and a collapsed node that has features displays a plus sign (+). A node that has no features has no sign.

The features of your selected working set classes available for selection are displayed in the Available Classes and Features list box at the left of the sheet. The icons at the left of features display a P, M, or C, indicating properties, methods, or constants, respectively. (You cannot export protected properties and methods.)

To include a feature in your exported package, perform one of the following actions:

- Double-click a class in the Available Classes and Features list box. The selected class and all of its features are then moved to the Selected Classes and Features list box.

- Select one or more of the features listed in the Available Classes and Features list box (by using the Shift or the Ctrl key to make multiple selections), and then click the single right-arrow (> button. Only the selected feature or features and the associated class or classes are moved to the Selected Classes and Features list box. You must repeat this action for each feature that you want to include in the working set.

- To select a class and all of its features, select the class in the Available Classes and Features list box and
then click the double right-arrow (\(>>\)) button. The selected class and all of its features are then automatically copied to the Selected Classes and Features list box.

You can select a feature once only. An attempt to select a feature that is already included in the working set has no effect.

The selected features and associated classes are automatically copied to the Selected Classes and Features list box and are displayed in green in the Available Classes and Features list box. Your selected features are displayed in the Selected Classes and Features list box class in the class in which they are defined. Your selections are not finalized until you click the Next > button. Before you click the Next > button, you can reverse any of your selections by removing them from the Selected Classes and Features list box.

To remove features from a class in the working set, perform one of the following actions

- Double-click a feature in the Selected Classes and Features list box.
- Select one or more of the features in the Selected Classes and Features list box (by using the Shift or the Ctrl key to make multiple selections) and then click the single left-arrow (\(<\)) button.
- To select all features in the Selected Classes and Features list box, click the double left-arrow (\(<<\)) button.

The selected features are then automatically moved from the Selected Classes and Features list box, click the double left-arrow (\(<<\)) button.

The selected features are displayed in green in the Available Classes and Features list box. Click the Next > button when you have selected all features that you require in the exported package working set. Alternatively, click the < Back button to redisplay the Create a Working Set of Classes sheet or the Cancel button to abandon your selections and close the wizard.

Selecting Interfaces

When you have selected the features for inclusion in the exported package working set and clicked the Next > button, the Select Interfaces for Package sheet of the Export Package Definition Wizard is then displayed, as shown in the following image.

![Select Interfaces for Package](image-url)
Use this sheet to select the interfaces that you want to include in the exported package.

The **Available Interfaces** list box contains interfaces defined in the current schema, which are all available for selection.

The **Selected Interfaces** list box contains the interfaces that you have selected for inclusion in your exported package. Interfaces that you select for inclusion are displayed in green in the **Available Interfaces** list box.

**To select the interfaces for your package, perform one of the following actions**

- Double-click an interface in the **Available Interfaces** list box.
- Select one or more of the interfaces listed in the **Available Interfaces** list box (by using the Shift or the Ctrl key to make multiple selections), and then click the single right-arrow (>) button.
- To select all interfaces from the **Available Interfaces** list box, click the double right-arrow (>>) button.

The selected interfaces are then automatically copied to the **Selected Interfaces** list box and are displayed in green in the **Available Interfaces** list box. You can select an interface once only. An attempt to select an interface that is already included has no effect.

Interfaces are added to the **Selected Interfaces** list box in alphabetical order. Your selections are not finalized until you click the Next > button. Before you click the Next > button, you can reverse any of your selections by removing them from the **Selected Interfaces** list box.

**To remove interfaces from your package, perform one of the following actions**

- Double-click an interface in the **Selected Interfaces** list box.
- Select one or more of the interfaces in the **Selected Interfaces** list box (by using the Shift or the Ctrl key to make multiple selections) and then click the single left-arrow (<) button.
- To select all interfaces in the **Selected Interfaces** list box, click the double left-arrow (<<) button.

The selected interfaces are then automatically moved from the **Selected Interfaces** list box to the **Available Interfaces** list box, where they are displayed in black.

Click the **Next >** button when you have selected all interfaces for inclusion in the package. Alternatively, click the **Back** button to redisplay the **Select Classes and Features for Package** sheet or the **Cancel** button to abandon your selections and close the wizard.
Changing Attributes

When you have selected the interfaces for inclusion in the exported package and clicked the Next > button, the Change Attributes sheet of the Export Package Definition Wizard is then displayed, as shown in the following image.

![Export Package Definition](image)

This sheet enables you to change selected attributes of any class or property in your exported package.

**To change the attributes of a class or property**

1. In the Package Components list box, select the class or property whose attributes you want to change. The attributes of that class or property are then displayed in the group box controls at the right of the sheet. Attributes that cannot be changed are disabled (for example, the class lifetimes or default persistence are disabled if you select a property or controls in the Access group box are disabled if you select a class).

2. Use the option buttons in the Classes or Properties group box to toggle the availability of classes or properties in the Package Components list box.

   By default, the selection of classes is enabled; that is, the Classes option button is selected. Select the Properties option button if you want to make the properties in your exported package working set available for selection.

3. If a property is selected in the Package Components list box and you want to change the access type for that property, select the ReadOnly or Public option button in the Access group box and then click the Update Access button. As you cannot export protected properties, this access type is not displayed.

   The access for that property in your working set is then changed and the icon at the left of the property in the Package Components list box displays the appropriate icon to reflect the public or read-only access type.
4. If a class is selected in the **Package Components** list box, check or uncheck the appropriate check boxes to specify the permitted lifetimes for instances of the class. By checking the appropriate check box, instances can have one or more of the following lifetimes.

- Persistent
- Shared transient
- Transient

For details, see "Specifying Class Lifetimes", in Chapter 3 of the JADE Development Environment User’s Guide. See also "Maintaining a Class in the Export Package Browser", later in this chapter.

5. If you want to change the default persistence of instances of the class selected in the **Package Components** list box, select the appropriate **Default Persistence** option button; that is, **Persistent** or **Transient**. You can still create a transient instance of a persistent class and a persistent instance of a transient class.

For details, see "Defining a Class", in Chapter 3 of the JADE Development Environment User’s Guide.

6. If a class is selected in the **Package Components** list box and you have changed the lifetime or persistence attributes of that class, click the **Update Lifetimes** button to save your changes.

7. Click the **Finish** button when you have changed all of the class and property attributes you require for components of your exported package.

Alternatively, click the `< Back` button to redisplay the **Select Classes and Features for Package** sheet to modify prior selections or the **Cancel** button to abandon your selections and close the wizard.

8. When you click the **Finish** button, if there are items referenced by your selection but not selected for inclusion, a Warning message box is then displayed, advising you of the items that are referenced by but not included in your selection, and allowing you to specify if you want to add them to your export package.

9. Click the **Yes** button to confirm that you want to include the specified items or the **No** button if you do not want to include them.

10. If you click the **Yes** button, the **Change Attributes** sheet is redisplayed with the newly included items displayed in blue.

    **Tip** You can click the `< Back button on the **Change Attributes** sheet to return previous sheets to refine your selection.

11. Click the **Finish** button when you have changed all of the class and property attributes you require for components of your exported package.

The Export Package Definition Wizard is then closed and your exported package created, based on your selections.

The exported package is then selected in the Export Packages Browser. You can now:

- Change the exported package definition (for details, see "Changing a Package by Using the Export Package Definition Wizard", in the following subsection)
- Extract it as a .scm file so that it can be imported for re-use by developers at another site or in another database (for details, see "Extracting a Package", later in this chapter)
- Imported into another schema on the same database (for details, see "Importing a Package", later in this chapter)
Changing a Package by Using the Export Package Definition Wizard

Use the Export Package Definition Wizard to maintain an existing exported package selected in the Export Packages Browser.

To change an exported package

1. Select the exported package in the Export Packages Browser.

2. Select the Change command from the Packages menu.

The Export Package Definition Wizard is then displayed. For details about the Export Package Definition Wizard steps, see "Using the Export Package Definition Wizard to Define a Package", earlier in this chapter.

When you are changing an existing exported package, the Package Name text box in the Define Package sheet displays the name of your selected exported package.

Tip You can also use the Export Package Browser to browse and maintain your exported packages. For details, see "Browsing and Maintaining an Exported Package", in the following section.

Browsing and Maintaining an Exported Package

You can browse and maintain an exported package by using the Export Package Browser instead of the Export Package Definition Wizard.

Tip The easiest way to maintain an exported package by using the Export Package Browser is to have an open Class Browser for the schema from which the exported package was defined and an open Export Package Browser positioned beside it.

You then simply drag and drop the classes and features that you want to include in your exported package from the Class Browser to the Export Package Browser and change the class and property attributes to meet your requirements.

To browse or maintain an exported package by using the Export Package Browser

1. Open a Class Browser for the schema in which the package is defined.

2. Open an Export Package Browser by performing the following actions.
   a. Select the Packages command from the Browse menu.
   b. Select the Export Package command from the submenu that is then displayed.
      The Export Packages Browser window is then opened, displaying all exported packages defined in that schema.
   c. Select the Browse command from the Packages menu. (The Browse command is disabled if the schema does not contain any exported packages.)
An Export Package Browser, shown in the following image, is then opened.

Use the Export Package Browser to browse the selected exported package as you would any other JADE entity (for example, the Primitive Types Browser or the Methods Browser).

For details about using the Export Package Browser to maintain attributes of classes and properties, see "Maintaining a Class in the Export Package Browser" or "Maintaining a Property in the Export Package Browser", later in this section.

3. To add a class or feature displayed in the Class Browser to the exported package displayed in the Export Package Browser, perform any of the following actions (noting that you cannot export protected properties and methods).

   - While holding down the Ctrl or the Shift key, select the class in the Class List of the Class Browser and drag the class to the Class List in the Export Package Browser.

     All properties, methods, and constants defined in that class are also copied to the Export Package Browser, where classes are displayed in green, indicating that they are part of the exported package working set.

     As only the selected class is copied, you must repeat this action for each subclass of the selected class that you want to include in the exported package working set.

   - While holding down the Ctrl or the Shift key, select the feature (property, method, or constant) from the appropriate list in the Class Browser and drag the feature to the Export Package Browser.

     Repeat this action for all features that you want to add to the working set.

   - Right-click in a list in the Export Package Browser and then select the Add command from the popup menu that is then displayed.
Alternatively, select the **Add** command from the Classes, Properties, Constants, or Methods menu accessed from the Export Package Browser.

The **Add** *entity-name* to Package dialog is then displayed. The dialog title reflects the type of entity that you want to add; for example, the Add Method to Package dialog.

The following image shows an example of the Add Class to Package dialog.

![Add Class to Package dialog](image)

All entities in the current schema are listed in the **Select Required Entry** list box, with those already included in the working set disabled from selection. Select the entity that you want to add to your exported package or enter the first character or the first few characters of the entity name in the **Find** text box to position that entity in the visible area of the **Select Required Entry** list box.

When you click the **OK** button, the selected entity is then added to the appropriate list in the Export Package Browser. (Alternatively, click the **Cancel** button to abandon your selection.)

Repeat this action for each entity you want to add by using the dialog. (Note that when you add a class by using this dialog, no features defined in that class are added to the exported package so you must individually add each feature of that class that you want to include.)

You can maintain attributes of a class or property included in the exported package working set from the Export Package Browser. For details, see "Maintaining a Class in the Export Package Browser" or "Maintaining a Property in the Export Package Browser", in the following subsections.

4. To add an interface to the exported package displayed in the Export Package Browser, right-click in the classes and interfaces list in the Export Package Browser and then select the **Add Interface** command from the popup menu that is then displayed.
The Add Interface to Package dialog is then displayed. All interfaces in the current schema are listed in the 
Select Required Entry list box, with those already included in the working set disabled from selection. 
Select the interface that you want to add to your exported package or enter the first character or the first few 
characters of the entity name in the Find text box to position that entity in the visible area of the Select 
Required Entry list box.

When you click the OK button, the selected entity is then added to the classes and interfaces list in the Export 
Package Browser. (Alternatively, click the Cancel button to abandon your selection.)

Repeat this action for each interface you want to add.

5. To remove a class or feature from the exported package working set, or to remove an interface, perform the 
following actions.

a. Select the item in the Export Package Browser that you want to remove.

b. Right-click on the item and then select the Remove command from the popup menu that is then 
displayed. Alternatively, select the Remove command from the Classes, Properties, Constants, or 
Methods menu, as appropriate.

The Confirm message box is then displayed, prompting you to confirm the deletion of the selected class or 
feature from the exported package working set.

Click the OK button to confirm the deletion or the Cancel button to cancel the deletion.

When you confirm the deletion, the selected item is then removed from the exported package and it is no 
longer displayed in the Export Package Browser.

**Note** If you selected a class for removal from the working set, all properties, methods, and constants 
defined in that class and previously included in the working set are also removed.

Repeat these actions for all classes, features, and interfaces that you want to remove from the package.

For details about locating a class in the current schema, see "Locating a Class", later in this section.

For details about exported packages, see "Exported Package Overview" under "Exported and Imported 
Packages", earlier in this chapter. For details about using a wizard to define or maintain an exported package, see 
"Exporting a Package Using the Export Package Definition Wizard", earlier in this chapter.

**Maintaining a Class in the Export Package Browser**

You can maintain the attributes of a class selected in the Class List of the Export Package Browser if you want 
attributes of a class in the exported package to differ from those of the class in the schema from which the package 
was built.

**To maintain a class in the exported package by using the Export Package Browser**

1. In the Class List of the Export Package Browser, right-click on the class whose attributes you want to view or 
change and then select the Change command from the popup menu that is then displayed. (Alternatively, 
select the class and then select the Change command from the Classes menu.)
The Exported Class Options dialog, shown in the following image, is then displayed.

![Exported Class Options dialog]

Attributes that cannot be changed are disabled.

2. In the Class Lifetimes group box, perform the following actions.

   a. If you do not want the class to have persistent instances, uncheck the **Allow Persistent Instances** check box. (This check box is disabled if the **Allow Persistent Subclass Instances** check box was unchecked in the superclass of this class.)

      When this is unchecked, an exception is raised at every attempt to create a persistent instance of the class. A compiler error is raised if you attempt to construct a persistent object from within a method (that is, you call `create <object-name> persistent`).

   b. If you do not want the class to have non-shared transient instances, uncheck the **Allow Transient Instances** check box. (The **Allow Transient Instances** check box is disabled if the **Allow Transient Subclass Instances** check box was unchecked in the superclass of this class.)

      When this is unchecked, an exception is raised at every attempt to create a non-shared transient instance of the class. A compiler error is raised if you attempt to construct a transient object from within a method (that is, you call `create <object-name> transient`).

   c. If you do not want the class to have shared transient instances, uncheck the **Allow Shared Transient Instances** check box. (The **Allow Shared Transient Instances** check box is disabled if the **Allow Shared Transient Subclass Instances** check box was unchecked in the superclass of this class.)

      When this is unchecked, an exception is raised at every attempt to create a shared transient instance of the class. A compiler error is raised if you attempt to construct a shared transient object from within a method (that is, you call `create <object-name> sharedTransient`).
3. Select the **Transient** option button in the Persistence group box if you do not want the default lifetime of instances of the class to be persistent. By default, the default lifetime of class instances is persistent. You can still create a transient instance of a persistent class and a persistent instance of a transient class.

Use the check boxes in the Class Lifetimes group box to restrict creation of class instances.

4. Click the **OK** button to update the class attributes. Alternatively, click the **Cancel** button to abandon your selections.

For details about using the Export Package Browser to view and maintain classes in an existing exported package, see "Browsing and Maintaining an Exported Package", earlier in this chapter.

**Maintaining a Property in the Export Package Browser**

You can maintain the attributes of a property selected in the Properties List of the Export Package Browser if you want attributes of a property in the exported package to differ from those of the property in the schema from which the package was built.

**To maintain a property in the exported package by using the Export Package Browser**

1. In the Properties List of the Export Package Browser, right-click on the property whose attributes you want to view or change and then select the **Change** command from the popup menu that is then displayed. (Alternatively, select the property and then select the **Change** command from the Properties menu.)

The Exported Property Options dialog is then displayed.

2. In the Access group box, select the appropriate type of access if you want to change the type of access to the property. As you cannot export protected properties, this access type is not displayed.

Select the **Public** option button if you want the property to be read and modified by any other method in the schema. (A property that has public access is displayed in the Properties List of the Export Package Browser with a spherical icon to the left of the property.)

Select the **ReadOnly** option button if you want the property to be read by other methods in the schema but not updated. (A read-only property is displayed in the Properties List of the Export Package Browser with the read-only spectacles icon to the left of the property.)

3. Click the **OK** button to update the property attributes. Alternatively, click the **Cancel** button to abandon your selections.

For details about using the Export Package Browser to view and maintain classes in an existing exported package, see "Browsing and Maintaining an Exported Package", earlier in this chapter.

**Locating a Class**

The **Find** command in the Export Package Browser is useful when you want to locate a class in the package and you have more classes in the exported package than are displayed in the Class List window.

**To locate a class in the Class List of the Export Package Browser**

1. Perform one of the following actions.
   - Press F4.
   - Right-click in the Class List and then select the **Find** command from the popup menu that is then displayed.
   - Select the **Find** command from the Classes menu.
The Find Type in Package dialog, shown in the following image, is then displayed.

All classes included in the exported package are listed in the Select Required Entry list box.

2. To select the class that you want to locate in the Class List of the Export Package Browser, perform one of the following actions.

   - Select the class that you require from the Select Required Entry list box.
     The classes included in the exported package working set are listed in alphabetical order.

   - Type at least the first few characters of the class name in the Find text box.
     The list display in the Select Required Entry list box starts with the class that matches your specified value. For example, if you have classes named ProcessStackArray, ProdMaint, Product, and ProductDict, if you enter pro in the Find text box, ProcessStackArray is then selected in the Select Required Entry list box. However, if you enter produ, the Product class is then selected.

   **Note** In the Export Package Browser, you cannot specify the class number in the Find text box as you can from the Class Browser.

3. When you have selected the required class, click the OK button. (Alternatively, click the Cancel button to abandon your search.)

   The selected class is then displayed in the Class List of the Export Package Browser.
Displaying References to an Exported Package

You can view the schema or schemas in your JADE database that reference (that is, import) a specific exported package.

To view schemas that reference an exported package

1. In the Export Packages Browser, select the exported package whose referencing schemas you want to view.
2. Select the References command from the Packages menu. A messages box is then displayed, listing the schemas that import the selected package.
3. When you have viewed the schema or schemas that import the selected exported package and the name or names with which the package was exported, click the OK button.

Importing a Package

The Import Packages Browser enables you to import and maintain a package in your current schema. Before you can import a package received from an exported package developer in another JADE database, you must first load the exported package schema into your JADE database.

When a package is imported, imported classes inherit from the local nearest RootSchema subschema class in the importing schema. A class can be imported only once into each schema branch.

While local (importing schema) classes can define references to imported classes, imported classes cannot participate in inverse relationships with locally defined classes.

A schema cannot import any packages exported from the same schema branch. In addition, you cannot subclass imported classes. For more details about imported packages, see "Imported Package Overview" under "Exported and Imported Packages", earlier in this chapter.

To open an Import Packages Browser

1. Select the Packages command from the Browse menu.
2. Select the Import Package command from the submenu that is then displayed.

The Import Packages Browser window is then opened. If you have not yet imported a package into the current schema, nothing is displayed in the Import Packages Browser.

Only one Import Packages Browser for the current schema can be open at any time. However, you can have concurrent open Import Packages Browsers for different schemas in a development work session.

You can change your default browser options, if required, by using the Browser sheet, accessed from the Options menu Preferences command.

Using the Packages Menu for Imported Packages

The Import Packages Browser provides the Packages menu, containing the commands listed in the following table, to enable you to import and maintain your imported packages.

<table>
<thead>
<tr>
<th>Command</th>
<th>For details, see...</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Adding an Imported Package to Your Schema</td>
<td>Displays the Import Package dialog</td>
</tr>
<tr>
<td>Change</td>
<td>Changing an Imported Package</td>
<td>Displays the Import Package dialog</td>
</tr>
</tbody>
</table>
Adding an Imported Package to Your Schema

The Import Packages Browser enables you to add one or more imported packages into the current schema.

**Note** Before you can import a package received from an exported package developer in another JADE database, you must first load the exported package schema into your JADE database.

**To add an imported package to the current schema**

1. If the Import Packages Browser does not have focus, select the **Import Package** command from the Packages submenu in the Browse menu.

2. Select the **Add** command from the Packages menu in the menu bar or the popup (context) menu when you right-click in the Import Packages Browser. (Alternatively, select the **Add** command from the Packages menu.)
3. In the Select Package To Import combo box, select the package that you want to import from the list of available packages in the list box portion of the control.

The list box not only lists all packages loaded into your database but at the first level in the hierarchy, the schema in which the package or packages were built. You can import packages at the second level (that is, schema-name:package-name); you cannot import the schema from which the package was built. Packages that have previously been imported into your schema or that are in the same schema branch are disabled.

4. As a package is imported with the export name by default, if you want to import the package with another name, specify the name for the imported package that is unique within the current schema (that is, it cannot be the same as that of another exported package or imported package in the schema).

The package name must start with an uppercase character and it cannot contain spaces.

5. In the Package Description text box, enter descriptive free-format text for your imported package, if required.

6. If you want to view the contents of the package, check the Show Details check box at the lower right of the dialog. (As package details are not displayed in this dialog by default, this check box is unchecked.)

An alphabetical list of all classes in the package and the features within each class are then displayed in the list box on the Package Contents sheet, for display purposes only.

**Tip** Bubble help provides details about a class or feature when the cursor is positioned over a class or feature in the Package Contents sheet list box for a few seconds.

7. To determine if the package conflicts in any way with the schema into which you are loading it, click the
Import Conflicts tab.

If no conflicts exist, the text *No class conflicts* is displayed. Any conflicts that exist are listed in the text box on the Import Conflicts sheet, as shown in the following example.

As a schema can contain a local class and an imported class with the same name, this conflict arises when an imported class of the stated name already exists in the schema.

8. Click the OK button to import the package. Alternatively, click the Cancel button to abandon the importation of the package.

The appropriate lists in the Class Browser are then update to show all imported classes and features in green.

**Changing an Imported Package**

Use the Import Package dialog to maintain information about an existing imported package selected in the Import Packages Browser. (You cannot change an imported constant.)

To change an imported package

1. Select the imported package in the Import Packages Browser.
2. Select the Change command from the Packages menu.

The Import Package dialog is then displayed. For details about using this dialog to import a package, see "Adding an Imported Package to Your Schema", earlier in this chapter. When you are changing an existing imported package, the Select Package To Import combo box is disabled.
Browsing and Maintaining an Imported Package

You can browse and maintain an existing imported package by using the Package Class Browser. You cannot use this browser to add or remove classes from the imported package but you can add features to an imported class, if required, by using standard JADE functionality. (For details, see "Defining Methods, Properties, Constants, and Conditions", in Chapter 4 of the JADE Development Environment User’s Guide.)

To browse or maintain an imported package by using the Package Class Browser

- Open a Package Class Browser by performing the following actions.
  a. Select the Packages command from the Browse menu.
  b. Select the Import Package command from the submenu that is then displayed. The Import Packages Browser window is then opened, displaying all packages imported into the schema.
  c. Select the Browse command from the Packages menu. (The Browse command is disabled if the schema does not contain any imported packages.)

A Package Class Browser is then opened, which enables you to browse the selected imported package as you would any other JADE entity (for example, the Primitive Types Browser or the Methods Browser).

You can add features (methods and constants) to an imported class if the attributes of that class permit.

Stub Packages

A stub package is an implementation of a package interface that supplies all of the classes, methods, and properties of the package but where each method is a stub that contains none of the functionality.

A stub package is useful in situations when a system needs to be deployed both with and without the full package being present. For example, the full package might be deployed only in sites that have paid an extra licence fee. Rather than deploying two versions of the system, you can write the importing schema as if the package is present and exported.

The stub version of the package means that the exporting classes, methods, and properties are present and can be compiled. The importing schema can then contain code like that shown in the following example.

```plaintext
optionallyCallPackage();
vars
  exported : Exported;
begin
  if Exported.PackagePresent then
    create exported transient;
    exported.exportedInt := 7;
    exported.exportedMethod(5,6);
  else
    write "Package not available";
  endif;
end;
```

The exported constant PackagePresent on the exported class Exported can be used to check whether the full package rather than the stub is present in the system and then act accordingly.
Although you can write a stub package from the definition of the package interface, it is difficult to do so, as it cannot be developed in the same system as the full package as it would conflict, as it has the same name. However, a simple approach is to develop the full schema with its full version of each package and then extract a version of the schema that contains only stub versions of each package. If this stub version of the schema is then deployed rather than the full version, the importing schema will compile without error and will be able to avoid using the functionality of the full schema as it can check that only the stub version is loaded.

Later, the full version of the schema can be deployed, which would make the full functionality available.

You can extract a stub version of the schema from the full version, by using the B batch extract argument in the jadclient non-GUI client application; for example:

```
jadclient path=c:\\jade\\system schema=JadeSchema app=JadeBatchExtract startAppParameters B c:\temp\\S1Stub.scm c:\temp\\scmS1.ddb S1
```

For details about batch extractions, see "Extracting Schemas as a Non-GUI Client Application", in Chapter 10 of the JADE Development Environment User's Guide.

This example extracts a stub version of the schema S1, containing only stub versions of any package to the files c:\temp\\S1Stub.scm and c:\temp\\S1Stub.ddb or c:\temp\\S1Stub.ddx. Any constant with the name PackagePresent on any exported class will be replaced by a Boolean constant with the value false.

In addition, the body of any exported method will be replaced by code that raises exception 1068 (Feature not available in this release) with error item text containing "Packages from schema S1 not available". This means that if the importing schema inadvertently calls an exported method stub (because it did not test the PackagePresent constant, for example), a suitable exception is raised.

**Extracting a Package**

You can extract an exported package or an imported package as part of the schema in which it is defined or imported. Alternatively, you can extract only the package itself. For more information about extracting and loading packages, see "Extracting and Loading Package Schemas", earlier in this chapter.

To extract an exported package or imported package only

1. In the Export Packages Browser or Import Packages Browser, select the package that you want to extract.
2. Select the Extract command from the Packages menu.
   The standard Save As dialog is then displayed.
3. Change the default file path and names, if required, and then click the OK button.

   By default, your package is extracted as .scm and .ddb or .ddx files prefixed with the name of your export or imported package; for example, DemoNotificationsPackage.scm. This file is extracted to your JADE working directory by default; for example:

   `s:\jade\test\bin`

For details about extracting an exported package as part of the schema in which it is defined or an imported package as part of the schema into which it is imported, see "Extracting Your Schema", in Chapter 10 of the JADE Development Environment User's Guide. However, you should first consider the caution under "Extracting and Loading Package Schemas", earlier in this chapter.

**Removing a Package**

You can physically delete a package from the Export Packages Browser or the Import Packages Browser.
To remove a package

1. In the Export Packages Browser or the Import Packages Browser, select the package that you want to physically delete from the JADE database.

2. Select the Remove command from the Packages menu. This command is disabled if you have selected an exported package in the Export Packages Browser that has been imported into one or more schemas in the JADE database. A Confirm message box is then displayed.

3. To confirm that you want to remove the imported package and all of its classes and features, click the OK button. Alternatively, click the Cancel button to abandon the deletion.

When the deletion of the package is successful, the browser is then updated to reflect the removal of the package.

Switching Application Contexts When Invoking a Method

When an imported method is invoked on an instance of an imported class or a class that implements an imported interface, the app, global, and currentSchema system variables switch to being those of the package. The package designer can rely on the appropriate application object being used in methods, depending on the context of the method.

- In the importing schema, app refers to the application object for the importing schema
- In the package schema, app refers to the application object for the package schema

This default rule for the objects referenced by the app, global, and currentSchema system variables when methods are invoked is exactly what is required in most situations. However, an extension to the normal syntax for invoking a method enables you to force a switch to a specific set of app, global, and currentSchema objects.

```java
// Normal syntax, which uses the default app, global, and currentSchema
obj.meth(params);

// Extension to use app, global, and currentSchema for a specific context
obj.meth(params) in (application context reference);
```

Notes  In the extended syntax, parentheses are required around the method parameters even if there are no actual parameters.

The compiler ensures that the parameters for the method to be invoked have the correct type and are in the correct order.

Parentheses are required around the application context reference unless it is a simple name.

The application context is obtained from the appContext system variable. In the following example, the importing schema obtains the application context of the package.

```java
vars
package : PackageClass;       // An imported class
context : ApplicationContext;
begin
   create package transient;
   context := package.getContext;       // Could save this value for future use
epilog
```
The `getContext` method returns the value of the `appContext` system variable within the package.

```plaintext
getContext() : ApplicationContext;
```

In the following example, a method written in the importing schema executes a method to write `currentSchema.name`, but by forcing a switch to the application context of the package, it is the name of the package schema that is output.

```plaintext
vars
    package : PackageClass;
begin
    create package transient;
    writeSchemaName;       // Importing schema name
    writeSchemaName() in (package.getContext); // Package schema name
  epilog
    delete package;
end;
```

### Calling User Methods from Packages

JADE enables you to provide an interface when creating a package to allow the user of the package to register a callback method to be invoked when a specific event occurs. (See also "Package Callback Examples", in the following subsection.)

When the package calls the method, it enables the schema in which the package is imported to perform the required functionality; for example, in an appointment book package in which you want to associate an alarm with an appointment.

When the starting time is reached, the alarm in the appointment book package calls the importing package callback method, which can send an e-mail message, bring up an alert form, or perform some other such action to alert the user of the impending appointment.

The package user must supply and the package writer must save the information required to perform the callback. A callback is registered by the package user supplying the target:

- **Object** (the receiver of the method call)
- **Method** (the method or condition to be invoked)
- **Context** (the context in which the method should be executed)

The package would normally export a method to allow the package user to supply this information.

Within a package, use the `Object` class `invokeMethod` method on the saved object to invoke the method and return the result. This method has as parameters the saved context and the saved method, and any parameters that are required.

```plaintext
invokeMethod(targetContext: ApplicationContext;
    targetMethod: Method;
    paramList: ParamListType): Any;
```

As part of the call, the `invokeMethod` call switches context to the saved context. This switch will typically be from the context of the package to the context of the user of the package.
Note As the application context used by `invokeMethod` is transient, it can switch to a context only within the same process. The mechanism is not designed to call a method running in another process in the node or in another node.

In addition, as the context is transient, any connection between a context and a method to be invoked must be set up again if an application is stopped and then restarted. If you want to save events to be called persistently so that methods would still be called if the application stops and restarts (for example, in a scheduler application), you would have to re-supply a context when the application restarts and events are loaded. The target method and object could be persistent but the context is not.

The `targetMethod` parameter must be a valid method, which is executed when the `invokeMethod` method is called. Use the `paramList` parameter to specify a variable list of parameters of any type that are passed to the method or condition specified in the `targetMethod` parameter when it is executed.

For details about the `ParamListType` pseudo type, see "ParamListType" under "Pseudo Types", in Chapter 1. See also "Passing Variable Parameters to Methods" under "JADE Language Syntax", in Chapter 1.

Notes If the number or type of the actual parameters passed to a method by a parameter list does not correspond exactly to the formal parameter list declaration, an exception or an unpredictable result may occur, as the compiler is unable to perform any type checking on the values that are passed to a parameter list. However, the `Method::isCallCompatibleWith` method, described earlier in this section, enables you to validate the number and type of parameters.

When an exception handler is invoked from a package method, the package context is switched to the context that was in effect when the exception handler was armed. When the exception handler finishes, the package context is adjusted back, if required.

Although the callback mechanism is designed with packages in mind, you can also use it to allow a method to be invoked from within the same context. If the context in the `invokeMethod` call is null, the current context (that is, `appContext`) is used. This therefore enables you to invoke a specific saved method (for example, `myClass::myMethod`) rather than calling the `Object::sendMsg` method, which allows you to provide only the name of the method to which the message is sent.

Within a package, the package writer may want to check that the method supplied by the user of the package is appropriate. The `Method::isCallCompatibleWith` method checks that the target method supplied by the package user cannot only be invoked on the specified target object but that it has a signature that is compatible with that expected by the package.

The `isCallCompatibleWith` method has the following signature.

```java
isCallCompatibleWith(targetObject: Object;
   exampleMethod: Method): Boolean;
```

When a process begins, transient instances of the `ApplicationContext` class are created (along with other environmental objects such as `app` and `global`) for the main application in which the package is imported and for each package application.

The `ApplicationContext` class provides the read-only properties listed in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains a reference to the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialApp</td>
<td><code>Application</code> instance of the context</td>
</tr>
<tr>
<td>initialPackage</td>
<td><code>JadePackage</code> instance of the context</td>
</tr>
<tr>
<td>initialProcess</td>
<td><code>Process</code> instance of the context; that is, the current process</td>
</tr>
<tr>
<td>initialSchema</td>
<td><code>Schema</code> instance of the context</td>
</tr>
</tbody>
</table>
To support the calling of user methods from packages, you can use the `appContext` system variable operand to denote the current context; that is, a transient instance of the `ApplicationContext` class. This instance encapsulates the current execution context and contains all of the information required for a method to switch back to the current context when a method is called. (Such a switch is the same as that which occurs when the context is switched on the call of an imported method.)

JADE automatically creates a unique `appContext` object for each JADE application that is running and for each package that is imported (recursively) by the schema from which that application is running.

**Note** The `appContext` system variable enables a package to switch context back to the context of the application that imported the package, by calling the `Object` class `invokeMethod` method.

### Package Callback Examples

The examples in this section (based on the Packages white paper on the JADE Web site at [https://www.jadeworld.com/developer-center/resource-library/white-papers](https://www.jadeworld.com/developer-center/resource-library/white-papers)) have a `ScheduledEvent` class in the exported package. This class has `targetMethod`, `targetContext`, and `targetObject` properties that are supplied to a `scheduleEvent` method. The `targetObject` property is an instance of any class in the schema using the package and the `targetMethod` property is a method on that class or a superclass of that class. The methods to be called in these examples are in the `DiaryTester` class (a subclass of `Application`), so the `app` system variable is used as the `targetObject` value.

The method in the following example schedules an event.

```plaintext
scheduleEvent_click(btn: Button input) updating;
begin
    scheduler.scheduleEvent(whenToInvoke, app,
        selectEvent.listObject.Method, appContext);
end;
```

The method to schedule an event can include a check that the target method supplied by the user of the package has the same signature as the `exampleCallback` method (for example, it may expect to call it with the scheduled time) and have the form: shown in the following example.

```plaintext
scheduleEvent(when: Time; targetObject: Object; targetMethod: Method; targetContext: ApplicationContext) updating;
vars
    se : ScheduledEvent;
begin
    if not targetMethod.isCallCompatibleWith(targetObject,
        Scheduler::exampleCallback) then
        write 'callback not valid';
        return;
    endif;
    create se transient;
    se.targetObject := targetObject;
    se.targetContext := targetContext;
    se.targetMethod := targetMethod;
    se.whenToStart := when;
    se.myScheduler := self;
    updateTimer;
end;
```
The method in the following example is called when the timer fires and inspects all events at the start of the queue and calls all those whose time has passed.

```plaintext
causeDueEvents();
vars
  se : ScheduledEvent;
begin
  foreach se in allScheduledEvents do
    if se.whenToStart > app.actualTime.time then
      return;
    endif;
    // Call users method, supplying expected start time as a parameter
    if se.targetObject <> null and se.targetMethod <> null then
      se.targetObject.invokeMethod(se.targetContext, se.targetMethod, se.whenToStart);
      se.myScheduler := null;
      delete se;
    endif;
  endforeach;
end;
```

**Summary of Restrictions when Using Packages**

When using packages, you should be aware of the following restrictions.

- You cannot export any of the following.
  - Imported classes.
  - Inherited properties and methods on a subclass. To export an inherited method, you must export the superclass or reimplement the inherited method on the exported class.
  - A RootSchema subschema copy class.
  - Subclasses of the Application, Global, or WebSession class.
  - Any system (that is, RootSchema) classes; for example, the Object or the Dictionary class.
  - Constructor (create) or destructor (delete) methods.
  - Event methods (for example, the timerEvent method, notification event methods, and control event methods), nor can you re-implement them in the importing schema.
  - Global constants.
  - External functions.
  - Properties that have protected access.

- As class names must be unique within a package, a single package cannot export both a class and a subschema copy of that class.

- Package names must be unique within a schema, across both imported and exported packages.

- A schema that exports a package cannot also import it. Schemas cannot import any packages that were exported from the same schema branch.

- Imported classes cannot participate in inverse relationships with local classes.

- You cannot subclass imported classes.
- A class can be imported once only, from one package only, within a specific schema branch. You cannot import a class more than once into a schema branch.
- You cannot remap imported classes to different database files.
- You cannot add local mapping methods to imported classes.
Chapter 9  Using Skins to Enhance JADE Applications

This chapter covers the following topics.

- Overview
- JADE Development Environment Skin Classes
  - JadeSkinRoot Class
  - JadeSkinEntity Class and Subclasses
- Run Time Skin Facilities
  - Defining Skin Category Names
  - Setting an Application Skin
  - Getting an Application Skin
  - Setting the Skin of a Form
  - Setting the Skin of a Control
  - Ignoring a Skin
- Extracting and Loading Skins
  - Using the JADE Skin Loader

Overview

A skin is a series of images that is applied to the caption line, menu line, and border areas of each form to provide an enhanced look and feel to each form. The skin can also define images for most controls to further enhance the look and feel of forms.

Each base control class with the ability to define a skin has an equivalent skin class associated with it. As these classes are defined in the RootSchema, skin data is global to all schemas. See also the JadeSkinRoot and JadeSkinEntity classes in Chapter 1 of the JADE Encyclopaedia of Classes and "Maintaining Skins Using Extended Functionality", in Chapter 2 of the JADE Runtime Application Guide.

An application can set a skin that is applied to all JADE forms or controls displayed during the running of that application in the current work session. User applications can enforce a specific skin (for example, a company logo) or they can provide the user with ability to select a preferred skin that is set during the application's initialize method (by calling the app.setSkin method).

JADE provides a collection of skins for the JADE development environment and a global collection that contains any user-defined skins for all schemas. As JADE must be able to upgrade existing systems by replacing the JADE system files, the skins used by the JADE development environment cannot be updated, and access to user-defined skins is not permitted.
Notes  Skins defined in the JADE development environment are stored in the _sysgui class map file and are therefore unavailable in user applications. Conversely, skins that you define for deployed applications are not available in the JADE development environment.

Before you can define a new skin for your applications, a picture file (for example, a .png, .bmp, .jpg, or .gif) must exist for each of the images that you want to specify.

A set of example skins is provided in the examples/skins directory. For details about loading these skins for use in your user application, see the examples/skins/readme.txt file in that directory.

JADE skins provide you with the following functionality.

- The ability to define a category both for a skin and for a window (form or control). Only those windows with the same category as the skin use that skin.
- The ability to provide an inactive skin for a form.
- Button images for the form close, max, and min buttons showing disabled, rollover, and down states.
- A global backColor property applied to all controls defined with their default three-dimensional color.
- The ability to skin almost all controls other than those that already cover the entire image with other drawing. Each skin can include the:
  - Type of border in terms of the border style or using eight images
  - Inner image, back brush, and background color
  - Foreground color
  - Font
  - Condition under which the skin is applied
- The ability to skin the menu line, drop-down menus, and popup (context) menus, as follows.
  - Type of border in terms of the borderStyle property or by using eight images
  - Inner image, backBrush, or backColor
  - backColor of selected items
  - Menu text foreColor, selected foreColor, and disabled foreColor
  - Font
  - Check box, separator, and right arrow images
- The ability to define the skin for an individual control.
- The ability to define the skin for a control class that is applied to the controls of that type at the application level.
- The ability to prevent a control or form from being skinned.

JADE Development Environment Skin Classes

In addition to the JadeSkin class, JADE provides classes that enable you to define a skin for each base control class. These classes are global to all of your schemas, as they are defined in the RootSchema.
**Note** As the *JadeSkin* class is completely separate from and does not provide the functionality provided by the classes in this chapter, it will be deimplemented in a future release.

You should therefore use the classes and features described in this section for your user applications.

With the exception of the *JadeSkin* class that is to be deimplemented in a later release, the JADE skin classes provided by the JADE *RootSchema* have the following hierarchy.

![Diagram of the JADE skin classes hierarchy]

These classes are summarized in the following subsections. For more details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.

For details about using runtime skin facilities provided by JADE methods, see "Run Time Skin Facilities", later in this chapter. See also "Maintaining Skins Using Extended Functionality", in Chapter 2 of the *JADE Runtime Application Guide*. 

---

**Chapter 9** Using Skins to Enhance JADE Applications

---

Developers Reference

---

242
The following `MemberKeyDictionary` subclasses enable you to access skin elements.

- JadeSkinApplicationNameDict
- JadeSkinCategoryNameDict
- JadeSkinControlNameDict
- JadeSkinEntityNameDict
- JadeSkinFormNameDict
- JadeSkinMenuNameDict
- JadeSkinSimpleButtonNameDict
- JadeSkinWindowStateNameDict

With the exception of the `JadeSkinEntityNameDict` class that allows duplicate keys, these dictionaries are defined with one key: `JadeSkinEntity::name`, which is case-sensitive and does not allow for duplicate keys. These dictionaries are referenced by the `JadeSkinRoot` class and are automatically maintained via inverses defined using the `JadeSkinEntity::mySkinRoot` property. In addition, each skin entity has a `JadeSkinEntityNameDict` dictionary of other skin entities that reference that skin. This `myOwners` dictionary is automatically maintained via inverses between the referencing property and the dictionary.

**JadeSkinRoot Class**

The `JadeSkinRoot` class, a subclass of the `Object` class, is the root class for all the skin entities. This class contains a series of dictionaries that enable you to reference the skin entities. Obtain the `JadeSkinRoot` class instance as follows.

```java
root := JadeSkinRoot.firstInstance;
```

The `JadeSkinRoot` class properties are automatically maintained member key dictionaries that use the `name` property of the `JadeSkinEntity` class as the key.

For details about the `JadeSkinRoot` class properties summarized in the following table, see Chapter 1 of the *JADE Encyclopaedia of Classes*.

<table>
<thead>
<tr>
<th>Property</th>
<th>Reference to the dictionary containing the name of all...</th>
</tr>
</thead>
<tbody>
<tr>
<td>allApplicationSkins</td>
<td>Application skins</td>
</tr>
<tr>
<td>allControlSkins</td>
<td>Control skins</td>
</tr>
<tr>
<td>allFormSkins</td>
<td>Form skins</td>
</tr>
<tr>
<td>allMenuSkins</td>
<td>Menu skins</td>
</tr>
<tr>
<td>allSimpleButtonSkins</td>
<td>Simple button skins</td>
</tr>
<tr>
<td>allSkinCategories</td>
<td>Skin categories</td>
</tr>
<tr>
<td>allSkinEntities</td>
<td>Skin entities</td>
</tr>
<tr>
<td>allWindowStateImages</td>
<td>Window state images</td>
</tr>
</tbody>
</table>

**JadeSkinEntity Class and Subclasses**

The `JadeSkinEntity` class, a subclass of the `Object` class, is the abstract superclass of the skin entities.
In addition to the `Default_Color` constant (whose default value means that a color is not set) provided by this class, the `JadeSkinEntity` class provides the properties summarized in the following table. For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>description</td>
<td>Text that can be used for documentation purposes</td>
</tr>
<tr>
<td>myOwners</td>
<td>Automatic collection of other skin entities that reference this object</td>
</tr>
<tr>
<td>mySkinRoot</td>
<td>Inverse reference to the instance of the <code>JadeSkinRoot</code> class</td>
</tr>
<tr>
<td>name</td>
<td>Name used to identify the skin entity</td>
</tr>
</tbody>
</table>

**JadeSkinApplication Class**

The `JadeSkinApplication` class, a subclass of the `JadeSkinEntity` class, defines an application skin.

An application skin definition consists of a collection of form and control skins. You can define a skin with no form skins (that is, with control skins only) and the reverse.

For details about the `JadeSkinApplication` class properties summarized in the following table, see Chapter 1 of the *JADE Encyclopaedia of Classes*.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Collection of ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>myFormSkins</td>
<td>Null</td>
<td>Form skins to be applied to the application</td>
</tr>
<tr>
<td>myControlSkins</td>
<td>Null</td>
<td>Control skins to be applied to the application</td>
</tr>
</tbody>
</table>

**JadeSkinArea Class**

The `JadeSkinArea` class, a subclass of the `JadeSkinEntity` class, defines the way in which a rectangular area is drawn.

The constants provided by the `JadeSkinArea` class are listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer Value</th>
<th>Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BorderStyle_3D</td>
<td>3</td>
<td>BorderStyle_3DSunken</td>
<td>2</td>
</tr>
<tr>
<td>BorderStyle_Images</td>
<td>4</td>
<td>BorderStyle_None</td>
<td>0</td>
</tr>
<tr>
<td>BorderStyle_Single</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For details about the `JadeSkinArea` class properties summarized in the following table, see Chapter 1 of the *JADE Encyclopaedia of Classes*.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>backColor</td>
<td>Default_Color</td>
<td>Background color of the area if the inner image is not supplied or it is not a brush</td>
</tr>
<tr>
<td>imgBorderBottomLeft</td>
<td>Null</td>
<td>Optional bottom left of the area</td>
</tr>
<tr>
<td>imgBorderBottomRight</td>
<td>Null</td>
<td>Optional bottom right of the area</td>
</tr>
<tr>
<td>imgBorderBottomStrip</td>
<td>Null</td>
<td>Optional bottom strip of the area</td>
</tr>
<tr>
<td>imgBorderLeftStrip</td>
<td>Null</td>
<td>Optional left strip of the area</td>
</tr>
</tbody>
</table>
The following image illustrates the layout of a skin, which is made up of eight border segments and an inner segment:

![Skin Layout Diagram]

A **JadeSkinArea** is drawn as follows.

- **Segment 1**
  
The top-left image (**imgBorderTopLeft**) is drawn at actual size. The top-left of the image is positioned at the top-left of the control.
  
The height drawn is usually the minimum of the top-left image height and the top-center strip height.
  
The exception is if the top-left image height is greater than the top-center strip height and the top-left image width is less than the left-center strip width. In that case, the top-left image height is used.
  
The top-left image can be higher than the top-center strip if the left-center strip is at least the same width.

- **Segment 2**
  
The top-right image (**imgBorderTopRight**) is drawn at actual size. The top-right of the image is positioned at the top-right of the control.
  
The height drawn is usually the minimum of the top-right image height and the top-center strip height.
  
The exception is if the top-right image height is greater than the top-center strip height and the top-right image width is less than the right-center strip width. In that case, the top-right image height is used.
  
The top-right image can be higher than the top-center strip if the right-center strip is at least the same width.

- **Segment 3**
  
The top center strip (**imgBorderTopStrip**) is drawn at actual image height and stretched horizontally between the top-left and top-right images.

- **Segment 4**
  
The bottom-left image (**imgBorderBottomLeft**) is drawn at actual size. The bottom-left of the image is positioned at the bottom-left of the control.
The height drawn is usually the minimum of the bottom-left image height and the bottom-center strip height.

The exception is if the bottom-left image height is greater than the bottom-center strip height and the bottom-left image width is less than the left-center strip width. In that case, the bottom-left image height is used.

This allows the bottom-left image to be higher than the bottom-center strip if the left-center strip is at least the same width.

- **Segment 5**

  The bottom-right image (imgBorderBottomRight) is drawn at actual size. The bottom-right of the image is positioned at the bottom-right of the control.

  The height drawn is usually the minimum of the bottom-right image height and the bottom-center strip height.

  The exception is if the bottom-right image height is greater than the bottom-center strip height and the bottom-right image width is less than the right-center strip width. In that case, the bottom-right image height is used.

  This allows the bottom-right image to be higher than bottom-center strip if the right-center strip is at least the same width.

- **Segment 6**

  The bottom center strip (imgBorderBottomStrip) is drawn at actual image height and stretched horizontally between the bottom-left and bottom-right images.

- **Segment 7**

  The left-center strip (imgBorderLeftStrip) is drawn at actual image width and stretched vertically between the top-left and the bottom-left images.

- **Segment 8**

  The right-center strip (imgBorderRightStrip) is drawn at actual image width and stretched vertically between the top-right and the bottom-right images.

- **Segment 9**

  The center image (imgInner) is drawn stretched from the left-center image to the right-center image and from the top-center image to the bottom-center image. If there is no center image, it is filled with the background color specified for the skin.

**Notes**  Segments 3 and 6 determine the respective top and bottom heights of the border. Segments 7 and 8 determine the respective left and right widths of the border.

Unexpected results may occur if an image has a size that is inappropriate or does not correspond to a specific area.

A corner segment is drawn to its full height if the width is the same as the corresponding left or right strip. For example, segment 1 can be higher that segment 3, provided that segment 1 is the same width as segment 7. You can use this to achieve rounded border effects. For details, see “JadeSkinWindow Class”, in the following section.

The following characteristics of the **JadeSkinArea** class are affected by additional subclass property values.

- If optional border images (that is, areas 1 through 8 in the above image) are not present, the inner area of the skin is the entire area.

- You can define the optional inner image by setting the value of the **imgInner** property to a brush that is repeatedly drawn over the entire inner area or an image that is drawn centered in the inner area.
A `backColor` property value is used only if the inner image (that is, the `imgInner` property) is not defined or it is not a brush.

For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*.

**JadeSkinMenu Class**

The *JadeSkinMenu* class, a subclass of the *JadeSkinArea* class, holds the skin drop down and popup menu definitions.

**Note** If the value of the `myMenuSkin` property for a form skin is null (a *JadeSkinMenu* reference), drop-down and popup menus are drawn using the skin menu color and font properties of the form, with a border style of `BorderStyle_3DRaised` (3).

For details about the *JadeSkinMenu* class properties summarized in the following table, see Chapter 1 of the *JADE Encyclopaedia of Classes*.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>backColorSelected</code></td>
<td>Default_Color</td>
<td>Contains the color used to draw the background of selected and enabled menu items in the drop-down or popup menu</td>
</tr>
<tr>
<td><code>borderStyle</code></td>
<td><code>BorderStyle_3DRaised</code></td>
<td>Contains the type of border to be drawn</td>
</tr>
<tr>
<td><code>fontBold</code></td>
<td>False</td>
<td>Specifies whether the menu font is bold</td>
</tr>
<tr>
<td><code>fontItalic</code></td>
<td>False</td>
<td>Specifies whether the menu font is italic</td>
</tr>
<tr>
<td><code>fontName</code></td>
<td>Null (&quot;&quot;)</td>
<td>Contains the name of the menu font</td>
</tr>
<tr>
<td><code>fontSize</code></td>
<td>0</td>
<td>Contains the size of the menu font</td>
</tr>
<tr>
<td><code>foreColor</code></td>
<td>Default_Color</td>
<td>Contains the color used to draw the text for non-selected and enabled menu items in the drop-down or popup menu</td>
</tr>
<tr>
<td><code>foreColorDisabled</code></td>
<td>Default_Color</td>
<td>Contains the color used to draw the text for disabled menu items in the drop-down or popup menu</td>
</tr>
<tr>
<td><code>foreColorSelected</code></td>
<td>Default_Color</td>
<td>Contains the color used to draw the text for selected menu items in the drop-down or popup menu</td>
</tr>
<tr>
<td><code>imgCheckMark</code></td>
<td>Null</td>
<td>Contains the image used to draw for checked menu items</td>
</tr>
<tr>
<td><code>imgRightArrow</code></td>
<td>Null</td>
<td>Contains the image used to draw the right arrow submenu indicator</td>
</tr>
<tr>
<td><code>imgSeparator</code></td>
<td>Null</td>
<td>Contains the image used to draw menu separators</td>
</tr>
<tr>
<td><code>lineHeight</code></td>
<td>0 (use Windows default)</td>
<td>Contains the height in pixels of each menu line item</td>
</tr>
<tr>
<td><code>pixelsAfterCheckMark</code></td>
<td>0</td>
<td>Contains the amount of space left after the checkmark column</td>
</tr>
<tr>
<td><code>pixelsAfterPicture</code></td>
<td>5</td>
<td>Contains the amount of space left after the picture image column</td>
</tr>
<tr>
<td><code>pixelsBeforeAccelerator</code></td>
<td>5</td>
<td>Contains the amount of space left before the accelerator text column</td>
</tr>
</tbody>
</table>
The `JadeSkinWindow` class, a subclass of the `JadeSkinArea` class, is the abstract superclass of all `Window` class skins and it contains the defined image and category of the skin. Applying a region to a window enables it to be of any shape and include “holes” inside it. In addition, any part of any child outside the defined region is not displayed.

For details about the `JadeSkinWindow` class properties summarized in the following table, see Chapter 1 of the JADE Encyclopaedia of Classes.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pixelsBeforeCheckMark</td>
<td>0</td>
<td>Contains the amount of space left before the checkmark column</td>
</tr>
<tr>
<td>pixelsBeforeRightArrow</td>
<td>0</td>
<td>Contains the amount space left before the right arrow column</td>
</tr>
</tbody>
</table>

To understand the way in which a region is constructed, consider that the image specified by the `myImageMask` property is drawn (stretched) over the top of the window, including any border area. Only black pixels are considered part of the window when it is painted or clicked on.

The result could be a window with rounded corners, a window with holes in it, and so on.

Because the mask image is built to the same size as the actual window, if border masks are defined, the only reasonable region that could be constructed is one where the corners are shaped. As all other areas are stretched to fit, it is likely that they would not provide a suitable result. You would achieve a better result by defining only an inner image for the mask that is stretched.

If the `mySkinCategory` property is set to "Company Logo", for example, the skin is applied only to a window of the appropriate type that has the `Window::skinCategoryName` property also set to "Company Logo".

In addition, you can define a skin category for a control subclass (for example, each `BaseControl` subclass) and associate a different category name with each of those skins. The constructor of each control subclass can then set the appropriate category name on the control so that the correct `JadeSkinBaseControl` class skin is then applied.

The `JadeSkinForm` class, a subclass of the `JadeSkinWindow` class, describes the skin of a form. Use the `JadeSkinArea` class to define the image drawn for the active border and the inner (client area) of the form. If the form has a backdrop picture, this image is drawn.

When erasing the inner area of the form:

1. If the form `backBrush` property is not null, the inner area is erased using that brush.
2. If the form skin has a `JadeSkinArea` class `imgInner` property value that is a brush, the inner area is erased using that brush.
3. If the form `backColor` property is not `Color_3DFace`, the inner area is erased using the background color of the form.
4. If the `BackColor` property of the skin area is not `Default_Color`, erase using the `BackColor` property value of the skin.

5. The `Color_3DFace` value is used when erasing.

6. If the form was erased using a color and the skin of the form has a `JadeSkinArea` class `imgInner` property value that is not a brush, that image is drawn centered in the inner area of the form skin.

The `JadeSkinForm` class contains the JADE skins defined for forms in your application and encapsulates the behavior required to define and maintain JADE skins using the `JadeSkinMaintenance` and `JadeSkinSelection` forms provided by the JADE RootSchema.

For details about using runtime skin facilities provided by JADE methods, see "Run Time Skin Facilities", later in this chapter.

The set of skin images used by JADE is provided with the product release so that you can use these skins in your applications, if required. (By default, skins are not used.)

The form border for a skin is made up of 11 images, as shown in the following image.

```
   1  2  3
4  5  6
7  8
9 10 11
```

The following is a description of the form border areas:

- Images 1, 3, 4, 6, 9, and 11 are shown at actual size.
- Images 2, 5, 7, 8, and 10 are stretched to fit the width or height of the form.
- Images 1, 2, and 3 must have the same height to enable the form to display correctly.
- Images 4, 5, and 6 must have the same height to enable the form to display correctly.
- Images 9, 10, and 11 must have the same height to enable the form to display correctly.
- The whole of image 1 is treated as the control menu area for the form.
- When an MDI child is maximized, the whole of image 4 is treated as the system menu area for the MDI child.
- Form icons are placed adjacent to the top right hand edge of the area defined by image 3.
- MDI child form icons are placed adjacent to the top right hand edge of the area defined by image 6.
- Form icons that are disabled are not displayed.

The following areas are not affected by using a skin.

- Only JADE forms adopt the skin presentation. Windows forms such as message boxes, common dialogs, and the JADE exception dialogs are unchanged.
- When a form is resized, Windows draws the standard form image while the resize is occurring.
- A minimized MDI form displays the standard image, as there is normally insufficient room to display the
skinned image for that short caption line.

- Windows-drawn menu items are unchanged by the skin. This includes the system menus.
- Any changes made to the skin do not affect any current users of that skin.
- Windows sounds do not occur when forms are minimized, maximized, and so on, as the form buttons are not in the Windows standard positions and their actions must be performed programmatically by JADE.
  (Windows does not issue those sounds when such actions are performed programmatically.)

For details about maintaining and using JADE skins, see "Defining and Maintaining JADE Skins at Run Time" under "Maintaining Skins Using Extended Functionality", in Chapter 2 of the JADE Runtime Application Guide.

For details about the JadeSkinForm class properties summarized in the following table, see Chapter 1 of the JADE Encyclopaedia of Classes.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>captionActiveForeColor</td>
<td>Default_Color</td>
<td>Color used to draw caption text when the form is active.</td>
</tr>
<tr>
<td>captionFontBold</td>
<td>False</td>
<td>Specifies whether the caption of the form is bold.</td>
</tr>
<tr>
<td>captionFontItalic</td>
<td>False</td>
<td>Specifies whether the caption of the form is italics.</td>
</tr>
<tr>
<td>captionFontName</td>
<td>Tahoma</td>
<td>Font with which the form caption is displayed.</td>
</tr>
<tr>
<td>captionFontSize</td>
<td>8.25</td>
<td>Size of the font with which the form caption is displayed.</td>
</tr>
<tr>
<td>captionInactiveForeColor</td>
<td>Default_Color</td>
<td>Color used to draw the form’s caption text when the form is inactive.</td>
</tr>
<tr>
<td>captionLeft</td>
<td>0</td>
<td>Starting left position of the form’s caption text.</td>
</tr>
<tr>
<td>captionTop</td>
<td>0</td>
<td>Starting right position of the form’s caption text.</td>
</tr>
<tr>
<td>centeredCaption</td>
<td>False</td>
<td>Specifies whether the caption is centered within the top strip area of the form’s skin.</td>
</tr>
<tr>
<td>imgInactiveBorderBottomLeft</td>
<td>Null</td>
<td>Border area image to be drawn for the bottom left of the inactive form.</td>
</tr>
<tr>
<td>imgInactiveBorderBottomRight</td>
<td>Null</td>
<td>Border area image to be drawn for the bottom right of the inactive form.</td>
</tr>
<tr>
<td>imgInactiveBorderBottomStrip</td>
<td>Null</td>
<td>Border area image to be drawn for the bottom strip of the inactive form.</td>
</tr>
<tr>
<td>imgInactiveBorderLeftStrip</td>
<td>Null</td>
<td>Border area image to be drawn for the left strip of the inactive form.</td>
</tr>
<tr>
<td>imgInactiveBorderRightStrip</td>
<td>Null</td>
<td>Border area image to be drawn for the right strip of the inactive form.</td>
</tr>
<tr>
<td>imgInactiveBorderTopLeft</td>
<td>Null</td>
<td>Border area image to be drawn for the top left of the inactive form.</td>
</tr>
<tr>
<td>imgInactiveBorderTopRight</td>
<td>Null</td>
<td>Border area image to be drawn for the top right of the inactive form.</td>
</tr>
<tr>
<td>imgInactiveBorderTopStrip</td>
<td>Null</td>
<td>Border area image to be drawn for the top strip of the inactive form.</td>
</tr>
<tr>
<td>Property</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>imgMenuLeft</td>
<td>Null</td>
<td>Image drawn for the left of the form’s menu line.</td>
</tr>
<tr>
<td>imgMenuRight</td>
<td>Null</td>
<td>Image drawn for the right of the form’s menu line.</td>
</tr>
<tr>
<td>imgMenuStrip</td>
<td>Null</td>
<td>Image drawn for the strip of the form’s menu line.</td>
</tr>
<tr>
<td>menuBackColor</td>
<td>Default_Color</td>
<td>Background color for the menu line if the skin has no defined imgMenuStrip property value.</td>
</tr>
<tr>
<td>menuBackColorSelected</td>
<td>Default_Color</td>
<td>Background color of the selected menu items for drop-down or popup menus.</td>
</tr>
<tr>
<td>menuFontBold</td>
<td>False</td>
<td>Specifies whether menu line item, drop-down menu, and popup menu captions are bold.</td>
</tr>
<tr>
<td>menuFontItalic</td>
<td>False</td>
<td>Specifies whether menu line item, drop-down menu, and popup menu captions are italics.</td>
</tr>
<tr>
<td>menuFontName</td>
<td>Null (&quot;&quot;)</td>
<td>Font with which menu line items, drop-down menus, and popup menus are displayed.</td>
</tr>
<tr>
<td>menuFontSize</td>
<td>0</td>
<td>Size of the font with which menu line items, drop-down menus, and popup menus are displayed.</td>
</tr>
<tr>
<td>menuForeColor</td>
<td>Default_Color</td>
<td>Color to draw the text for non-selected and enabled menu line items.</td>
</tr>
<tr>
<td>menuForeColorDisabled</td>
<td>Default_Color</td>
<td>Color to draw the text for disabled menu line items and the default disabled text color for drop-down and popup menus.</td>
</tr>
<tr>
<td>menuForeColorSelected</td>
<td>Default_Color</td>
<td>Color to draw text for selected text for drop-down and popup menus.</td>
</tr>
<tr>
<td>menuLeftPosition</td>
<td>0</td>
<td>Starting left position of the form’s menus.</td>
</tr>
<tr>
<td>menuTopPosition</td>
<td>0</td>
<td>Offset of the top position of the form’s menu drawn on the skin.</td>
</tr>
<tr>
<td>myChildMinimizeBtn</td>
<td></td>
<td>Reference to the simple button images drawn for an MDI child minimize button in its four states (up, down, rollover, and disabled).</td>
</tr>
<tr>
<td>myChildRestoreBtn</td>
<td></td>
<td>Reference to the simple button images drawn for an MDI child restore button in its four states.</td>
</tr>
<tr>
<td>myChildTerminateBtn</td>
<td></td>
<td>Reference to the simple button images drawn for an MDI child terminate button in its four states.</td>
</tr>
<tr>
<td>myMaximizeBtn</td>
<td></td>
<td>Reference to the simple button images drawn for an MDI child maximize button in its four states.</td>
</tr>
<tr>
<td>myMaximizedBtn</td>
<td></td>
<td>Reference to the simple button images drawn for an MDI child maximized button in its four states.</td>
</tr>
<tr>
<td>myMenuSkin</td>
<td></td>
<td>Reference to the menu definition of the form.</td>
</tr>
<tr>
<td>myMinimizeBtn</td>
<td></td>
<td>Reference to the simple button images to be drawn for the form minimize button in its four states.</td>
</tr>
<tr>
<td>myTerminateBtn</td>
<td></td>
<td>Reference to the simple button images to be drawn for the form terminate button in its four states.</td>
</tr>
</tbody>
</table>
### JadeSkinControl Class

The JadeSkinControl class, a subclass of the JadeSkinWindow class, is the abstract superclass that provides the definition of elements common for each control that can be skinned. For details, see Chapter 1 of the JADE Encyclopaedia of Classes.

Use the properties of the JadeSkinArea class to define the image drawn for the active border and the inner (client area) of each control. When erasing the inner area of a control:

1. If the control backBrush property of the form is not `null`, the inner area is erased using that brush.
2. If the control skin has a JadeSkinArea class imgInner property value that is a brush, the inner area is erased using that brush.
3. If the backColor property of the skin area is not Default_Color and the backColor property of the control is the default or the skin was set by using the Control class setSkin method, erase using the backColor property value of the skin.
4. If the value of the backColor property of the control is Color_3DFace and the form of the control has a skin whose backColor property value is not set to Default_Color, the inner area is erased using the backColor property value of the form’s skin.
5. Erase using the backColor property value of the control.
6. If the control was erased using a color and the skin of the control has an inner image defined in the JadeSkinArea class imgInner property that is not a brush (that is, the JadeSkinArea class innerIsBrush property is set to `false`), that image is drawn centered in the inner area.

The constants provided by the JadeSkinControl class are listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApplyCondition_3D</td>
<td>2</td>
<td>ApplyCondition_All</td>
<td>0</td>
</tr>
<tr>
<td>ApplyCondition_Border</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For details about the JadeSkinControl class properties summarized in the following table, see Chapter 1 of the JADE Encyclopaedia of Classes.

<table>
<thead>
<tr>
<th>Property</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applyCondition</td>
<td>Dependent on the control type</td>
<td>Determines whether the border area of a control uses the skin</td>
</tr>
<tr>
<td>borderStyle</td>
<td>BorderStyle_Images</td>
<td>Type of border to be drawn</td>
</tr>
<tr>
<td>fontBold</td>
<td>False</td>
<td>Specifies whether the control font is bold when the control uses the default application font</td>
</tr>
<tr>
<td>fontItalic</td>
<td>False</td>
<td>Specifies whether the control font is italicized when the control uses the default application font</td>
</tr>
<tr>
<td>Property</td>
<td>Default Value</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>fontName</td>
<td>Null (&quot;&quot;)</td>
<td>Font with which the control is drawn (the default value indicates the control uses its own default font)</td>
</tr>
<tr>
<td>fontSize</td>
<td>0</td>
<td>Specifies the size of the control font when the control uses the default application font</td>
</tr>
<tr>
<td>fontStrikethru</td>
<td>False</td>
<td>Specifies whether the control font is strikethrough when the control uses the default application font</td>
</tr>
<tr>
<td>fontUnderline</td>
<td>False</td>
<td>Specifies whether the control font is underlined when the control uses the default application font</td>
</tr>
<tr>
<td>foreColor</td>
<td>Default_Color</td>
<td>Color to be used for drawing the text of the control when it is disabled</td>
</tr>
<tr>
<td>foreColorDisabled</td>
<td>Default_Color</td>
<td>Color to be used for drawing the text of the control</td>
</tr>
</tbody>
</table>

The subclasses of the JadeSkinControl class are summarized in the following table. For details about these subclasses and their properties, see Chapter 1 of the JADE Encyclopaedia of Classes.

<table>
<thead>
<tr>
<th>Subclass</th>
<th>Holds the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>JadeSkinBaseControl</td>
<td>Definition of a skin for BaseControl subclassed controls</td>
</tr>
<tr>
<td>JadeSkinBrowseButtons</td>
<td>Definition of a skin for BrowseButtons controls</td>
</tr>
<tr>
<td>JadeSkinButton</td>
<td>Definition of a skin for Button controls</td>
</tr>
<tr>
<td>JadeSkinCheckBox</td>
<td>Definition of a skin for CheckBox controls</td>
</tr>
<tr>
<td>JadeSkinComboBox</td>
<td>Definition of a skin for ComboBox controls</td>
</tr>
<tr>
<td>JadeSkinFolder</td>
<td>Definition of a skin for Folder controls</td>
</tr>
<tr>
<td>JadeSkinFrame</td>
<td>Definition of a skin for Frame controls</td>
</tr>
<tr>
<td>JadeSkinGroupBox</td>
<td>Definition of a skin for GroupBox controls</td>
</tr>
<tr>
<td>JadeSkinJadeDockBase</td>
<td>Elements of a docking control skin and provides the JadeSkinJadeDockBar and JadeSkinJadeDockContainer subclasses that hold definitions of skins for the JadeDockBar and JadeDockContainer classes</td>
</tr>
<tr>
<td>JadeSkinJadeEditMask</td>
<td>Definition of a skin for JadeEditMask controls</td>
</tr>
<tr>
<td>JadeSkinJadeMask</td>
<td>Definition of a skin for JadeMask controls</td>
</tr>
<tr>
<td>JadeSkinJadeRichText</td>
<td>Definition of a skin for JadeRichText controls</td>
</tr>
<tr>
<td>JadeSkinLabel</td>
<td>Definition of a skin for Label controls</td>
</tr>
<tr>
<td>JadeSkinListBox</td>
<td>Definition of a skin for ListBox controls</td>
</tr>
<tr>
<td>JadeSkinOleControl</td>
<td>Definition of a skin for OleControl controls</td>
</tr>
<tr>
<td>JadeSkinOptionButton</td>
<td>Definition of a skin for OptionButton controls</td>
</tr>
<tr>
<td>JadeSkinPicture</td>
<td>Definition of a skin for Picture controls</td>
</tr>
<tr>
<td>JadeSkinProgressBar</td>
<td>Definition of a skin for ProgressBar controls</td>
</tr>
<tr>
<td>JadeSkinScrollBar</td>
<td>Information common to a vertical and horizontal scroll bar, and provides the JadeSkinHScroll and JadeSkinVScroll subclasses that hold definitions of skins for the HScroll and VScroll classes</td>
</tr>
<tr>
<td>JadeSkinHScroll</td>
<td>Definition of a skin for HScroll horizontal ScrollBar controls</td>
</tr>
<tr>
<td>Subclass</td>
<td>Holds the …</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>JadeSkinVScroll</td>
<td>Definition of a skin for ScrollBar vertical ScrollBar controls</td>
</tr>
<tr>
<td>JadeSkinSheet</td>
<td>Definition of a skin for Sheet controls</td>
</tr>
<tr>
<td>JadeSkinStatusLine</td>
<td>Definition of a skin for StatusLine controls</td>
</tr>
<tr>
<td>JadeSkinTable</td>
<td>Definition of a skin for Table controls</td>
</tr>
<tr>
<td>JadeSkinTextBox</td>
<td>Definition of a skin for TextBox controls</td>
</tr>
</tbody>
</table>

Skins do not apply to the ActiveXControl, MultiMedia, and Ocx control classes, as these are totally drawn by the controls themselves.

**JadeSkinBaseControl Class**

The JadeSkinBaseControl class, a subclass of the JadeSkinControl class, holds the definition of a skin for subclasses of the BaseControl class.

**JadeSkinBrowseButtons Class**

The JadeSkinBrowseButtons class, a subclass of the JadeSkinControl class, holds the definition of a skin for BrowseButtons controls.

If a button image is not supplied for a non-up state, the up image is used. For painting to be successful, the skin requires all of the up images to be supplied.

**JadeSkinButton Class**

The JadeSkinButton class, a subclass of the JadeSkinControl class, holds the definition of a skin for Button controls.

If the createRegionFromMask property is set to true, the JadeSkinWindow::myImageMask property is used to create a window region. If it is set to false (the default), the full rectangular button area is drawn using the skin. The region defined by the myImageMask property then applies only to mouse actions. For example, if the button is an unusual-shaped image on a background, the button displays only the rollover and click images when the mouse is positioned over that special area.

Each state can consist of up to eight border segments and an inner image, an inner image only, or no images (in which case the background color is used to fill the non-border area). These images are drawn inside any defined border area. If you do not define a specific state, the myButtonUp image is used.

The following image on the left is an example of a button with a raised three-dimensional effect and the image on the right is an example of a button with a sunken three-dimensional effect.

**JadeSkinCheckBox Class**

The JadeSkinCheckBox class, a subclass of the JadeSkinControl class, holds the definition of a skin for CheckBox controls. If the check box button image of the skin is higher than the check box control using that skin, the check box control is enlarged in height to display the entire button image.
If you do not supply a specific state, the appropriate up image is used. If you do not supply the appropriate up image, the default check box image is drawn. For example, the following image is an example of a CheckBox control with the Control class borderStyle property set to BorderStyle_3DSunken and the Control class backBrush property set.

![Checkbox Image](image)

**JadeSkinComboBox Class**

The JadeSkinComboBox class, a subclass of the JadeSkinControl class, holds the definition of a skin for ComboBox controls.

When defining the skin for a ComboBox control, note the following points.

- A simple combo box is a text box followed by a list box covering the whole combo area.
- A spin box ignores any defined buttons for the combo box skin. The skin of the vertical scroll bar for the application is used to draw the spin box over the top of any defined border.
- When the combo box has a text box portion, the text box allows only a solid background color and does not successfully handle any defined brush.
- A combo box skin consists of a border definition and a button that is placed on the right side of the border area. The way in which it is painted depends on the defined border, as follows.
  - If the combo box skin has the JadeSkinControl class borderStyle property set to BorderStyle_UseImages, the button image is centered vertically and offset from the right-hand edge of the combo box by the value of the buttonRightOffset property, as shown in the following example.

![ComboBox Image](image)

  - If the combo box skin has the borderStyle property set to a value other than BorderStyle_UseImages, the button is drawn inside whatever border is defined (against right inner edge of the border and stretched vertically), as shown in the following example.

![ComboBox Image](image)

**JadeSkinFolder Class**

The JadeSkinFolder class, a subclass of the JadeSkinControl class, holds the definition of a skin for Folder controls.

**JadeSkinFrame Class**

The JadeSkinFrame class, a subclass of the JadeSkinControl class, holds the definition of a skin for Frame controls.

**JadeSkinGroupBox Class**

The JadeSkinGroupBox class, a subclass of the JadeSkinControl class, holds the definition of a skin for GroupBox controls. As a group box control has no non-client area (border area) and the entire skin image is drawn in the client area, children may be positioned anywhere within that control and cover the drawn images.
The group box skin images are drawn over the entire control area. The drawing of the caption depends on whether the skin definition includes a skin label reference (myLabelSkin). If there is no skin label reference, the caption is drawn transparently over the skin image. If the skin references a label definition, that label skin is drawn on top of the group box skin image.

The label is sized so that there is a three-pixel gap from the border area of the label to the left, right, top, and bottom of the caption. The caption or the label is drawn at the position indicated by the captionPosition, captionPositionLeftOffset, and captionPositionTopOffset properties.

The constants provided by the JadeSkinGroupBox class are listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CaptionPosition_Left_Top</td>
<td>0</td>
<td>Left-justified at the top</td>
</tr>
<tr>
<td>CaptionPosition_Left_Middle</td>
<td>1</td>
<td>Left-justified and centered vertically</td>
</tr>
<tr>
<td>CaptionPosition_Left_Bottom</td>
<td>2</td>
<td>Left-justified at the bottom</td>
</tr>
<tr>
<td>CaptionPosition_Right_Top</td>
<td>3</td>
<td>Right-justified at the top</td>
</tr>
<tr>
<td>CaptionPosition_Right_Middle</td>
<td>4</td>
<td>Right-justified and centered vertically</td>
</tr>
<tr>
<td>CaptionPosition_Right_Bottom</td>
<td>5</td>
<td>Right-justified at the bottom</td>
</tr>
<tr>
<td>CaptionPosition_Center_Top</td>
<td>6</td>
<td>Centered horizontally at the top</td>
</tr>
<tr>
<td>CaptionPosition_Center_Middle</td>
<td>7</td>
<td>Centered horizontally and vertically</td>
</tr>
<tr>
<td>CaptionPosition_Center_Bottom</td>
<td>8</td>
<td>Centered horizontally at the bottom</td>
</tr>
</tbody>
</table>

The following image is an example of two group boxes. The example at the left has only the top and right border strips set.

![GroupBox](image)

**JadeSkinJadeDockBase Class**

The JadeSkinJadeDockBase class, a subclass of the JadeSkinControl class, is the abstract class that defines elements of a docking control skin.

The first example at the left of the following image shows an image drawn horizontally (by using the myHorizontalGripBar property) for the grip bar of a vertically aligned docking control, in which the grip bar is the two horizontal lines at the top of the image. The second example at the right of the following image shows an image drawn vertically (by using the myVerticalGripBar property) for the grip bar for a horizontally aligned docking control, in which the grip bar is the two vertical lines at the left of the image.

![Docking Control](image)

The JadeSkinJadeDockBase class provides the JadeSkinJadeDockBar control subclass and JadeSkinJadeDockContainer control subclass.
JadeSkinJadeDockBar Class
The **JadeSkinJadeDockBar** class, a subclass of the **JadeSkinJadeDockBase** class, holds the definition of a skin for **JadeDockBar** controls.

JadeSkinJadeDockContainer Class
The **JadeSkinJadeDockContainer** class, a subclass of the **JadeSkinJadeDockBase** class, holds the definition of a skin for **JadeDockContainer** controls.

JadeSkinJadeEditMask Class
The **JadeSkinJadeEditMask** class, a subclass of the **JadeSkinControl** class, holds the definition of a skin for **JadeEditMask** controls.

**Note** The background area of a **JadeEditMask** control outside of the text box children is always drawn using the value of the **backColor** property of its parent and is unaffected by the skin.

JadeSkinJadeMask Class
The **JadeSkinJadeMask** class, a subclass of the **JadeSkinControl** class, holds the definition of a skin for **JadeMask** controls.

JadeSkinJadeRichText Class
The **JadeSkinJadeRichText** class, a subclass of the **JadeSkinControl** class, holds the definition of a skin for **JadeRichText** controls.

JadeSkinLabel Class
The **JadeSkinLabel** class, a subclass of the **JadeSkinControl** class, holds the definition of a skin for **Label** controls.

JadeSkinListBox Class
The **JadeSkinListBox** class, a subclass of the **JadeSkinControl** class, holds the definition of a skin for **ListBox** controls.

**Note** The images defined in these properties replace the equivalent image only if the standard JADE image has not been replaced in the list box.

The following image shows an example of a **ListBox** class with a defined skin.
JadeSkinOleControl Class

The JadeSkinOleControl class, a subclass of the JadeSkinControl class, holds the definition of a skin for OleControl controls.

JadeSkinOptionButton Class

The JadeSkinOptionButton class, a subclass of the JadeSkinControl class, holds the definition of a skin for OptionButton controls. If the option button image of the skin is higher than the option button control using that skin, the option button control is enlarged in height to display the entire image.

If you do not supply a specific state, the appropriate up image is used. If you do not supply the appropriate up image, the default option button image is drawn.

JadeSkinPicture Class

The JadeSkinPicture class, a subclass of the JadeSkinControl class, holds the definition of a skin for Picture controls.

JadeSkinProgressBar Class

The JadeSkinProgressBar class, a subclass of the JadeSkinControl class, holds the definition of a skin for ProgressBar controls.

JadeSkinScrollBar Class

The JadeSkinScrollBar class, a subclass of the JadeSkinControl class, is the abstract class that contains information common to vertical and horizontal scroll bars.

The JadeSkinScrollBar class provides the JadeSkinHScroll and JadeSkinVScroll control subclasses.

JadeSkinHScroll Class

The JadeSkinHScroll class, a subclass of the JadeSkinScrollBar class, holds the definition of a skin for HScroll subclasses of ScrollBar controls.

Note  This skin is also used for drawing the horizontal scroll bar for any control or form in the application.

The height of the border area and the height of the myLeftButton.imgUp image determine the height of the scroll bar. For a scroll bar control, the image is stretched vertically to fit the area inside the borders, as shown in the example in the following image.

If you do not supply the image for a specific state, the appropriate up image is used. The default scroll button image is drawn if you do not supply the up image.

JadeSkinVScroll Class

The JadeSkinVScroll class, a subclass of the JadeSkinScrollBar class, holds the definition of a skin for VScroll subclasses of ScrollBar controls.

Note  This skin is also used for drawing the vertical scroll bar for any control or form in the application.

The width of the border area and the width of the myTopButton.imgUp image determine the width of the scroll bar. For a scroll bar control, the image is stretched horizontally to fit the area inside the borders.
If you do not supply the image for a specific state, the appropriate up image is used. The default scroll button image is drawn if you do not supply the up image.

**JadeSkinSheet Class**

The *JadeSkinSheet* class, a subclass of the *JadeSkinControl* class, holds the definition of a skin for *Sheet* controls.

The following image shows an example of a sheet with a sunken three-dimensional effect.

![Sheet Example](image)

**JadeSkinStatusLine Class**

The *JadeSkinStatusLine* class, a subclass of the *JadeSkinControl* class, holds the definition of a skin for *StatusLine* controls.

The following image shows an example of a skinned status line.

![StatusLine Example](image)

**JadeSkinTable Class**

The *JadeSkinTable* class, a subclass of the *JadeSkinControl* class, holds the definition of a skin for *Table* controls.

The default value of *Default_Color* for these properties indicates that the table uses its normal color. The following image shows an example of the *Table* class with an applied skin.

![Table Example](image)

**JadeSkinTextBox Class**

The *JadeSkinTextBox* class, a subclass of the *JadeSkinControl* class, holds the definition of a skin for *TextBox* controls.

**Note** Text box controls do not successfully handle back brushes and non-solid colors for the *backColor* property.
JadeSkinWindowStatelmage Class

The JadeSkinWindowStatelmage class, a subclass of the JadeSkinArea class, defines the image of a window area for a specific state (that is, up, down, rollover, or disabled). For details about the JadeSkinWindowStatelmage class, see Chapter 1 of the JADE Encyclopaedia of Classes.

JadeSkinCategory Class

The JadeSkinCategory class, a subclass of the JadeSkinEntity class, holds the skin category definitions.

JadeSkinSimpleButton Class

The JadeSkinSimpleButton class, a subclass of the JadeSkinEntity class, holds the skin definitions for a simple button and its up, down, disabled, and rollover states.

Each state is defined by using a single image. Simple buttons are used to define the form buttons such as the Maximize and Minimize buttons and the buttons for a check box, option button, scroll bar, and combo box.

For details about the JadeSkinSimpleButton class, see Chapter 1 of the JADE Encyclopaedia of Classes.

Run Time Skin Facilities

JADE provides the following facilities that are available to any developer.

Skin Facilities Available Prior to JADE 6.0

The following skin facilities were available in JADE 5.1 and 5.2 releases.

- **Application** class getSkin method, which returns a reference to the JadeSkin object that is currently set. If no skin is currently set, a null value is returned.
- **Application** class getSkinCollection method, which returns the global collection of skins. This collection is global to all schemas and is automatically created by the first app.getSkinCollection call. (See also “Getting an Application Skin”, later in this chapter.) To implement your own selection facility, display the name property for each JadeSkinEntity or JadeSkin object in the collection.
- **Application** class setSkin method, which defines the skin to be used by the application by setting the skin that applies to the application that is currently running. To cancel skin usage for the application, pass null as the skin object; that is, app.setSkin(null).

New Skin Facilities in JADE 6.0 and Higher

The following skin facilities were implemented in the JADE 6.0 release.

- **JadeSkinMaintenance** form, which enables you to define and maintain skins for user applications at runtime. For details, see "Defining and Maintaining JADE Skins at Runtime", in Chapter 2 of the JADE Runtime Application Guide.
- **JadeSkinSelection** form, which enables a user to select or cancel the skin to use in the application at runtime. For details, see "Selecting a Skin to Use in Runtime Applications", in Chapter 2 of the JADE Runtime Application Guide.
- **JadeSkinRoot** class, containing a series of dictionaries by which the various skin entities can be referenced. Use the Class::firstInstance method to obtain the class instance, as follows.

    root := JadeSkinRoot.firstInstance;
- **Application** class `getApplicationSkin` method, which returns the `JadeSkinApplication` object that is currently set for the application. If no skin is currently set, a `null` value is returned. For details, see "Getting an Application Skin", later in this chapter.

- **Application** class `setApplicationSkin` method, which sets the skin for the current application. For details, see "Setting an Application Skin", later in this chapter.

- **Window** class `skinCategoryName` property, containing the skin category name defined of each form and control. For details, see "Defining Skin Category Names", later in this chapter.

- **Control** class `setSkin` method, which sets the skin for a specific control regardless of the setting of the **Window**: `skinCategoryName` property. For details, see "Setting the Skin of a Control", later in this chapter.

- **Form** class `setFormSkin` method, which sets the skin for a specific form regardless of the setting of the **Window**: `skinCategoryName` property. For details, see "Setting the Skin of a Form", later in this chapter.

- **Form** class `setApplicationSkin` method, which sets the skin for a specific form and its controls. For details, see "Setting the Skin of a Form", later in this chapter.

- **Window** class `ignoreSkin` property, which specifies whether a window uses a skin. For details, see "Ignoring a Skin", later in this chapter.

### Defining Skin Category Names

The **Window** class `skinCategoryName` property enables you to define each form and control with a skin category. This property does not contain a reference but a string name that can be read or written to at any time. By default, the value of this property is null (""").

When an application skin is set, the window uses only a skin that matches the defined category. As the linkage between the window and the skin category is deliberately flexible, you can replace or ignore skins to meet your requirements.

### Setting an Application Skin

The **Application** class `setApplicationSkin` method, which has the following signature, sets the skin for the current application.

```java
setApplicationSkin(skin: JadeSkinApplication);
```

The skin consists of a collection of form and control skins. (One of the collections may be empty.) When you call this method to set a skin, any existing skin for the application is replaced.

When the **Window** class `ignoreSkin` property is set to false, each form that is active or subsequently loaded applies the form skin that matches the value of the `skinCategoryName` for the form. If no matching form skin is found, a skin is not applied to that form.

Each control that is active or subsequently loaded applies the corresponding control skin matching the value of the `skinCategoryName` property. If there is no matching control skin or the value of the `ignoreSkin` property is `true`, the control is not drawn with a skin.

**Note** Changing a skin object after this method is called has no impact on the displayed skin. To apply any skin changes dynamically, you must call `app.setApplicationSkin` again.

To clear the application skin (cancel the skin display), call this method again with a null value, as follows.

```java
app.setApplicationSkin(null);
```
Getting an Application Skin

The Application class getApplicationSkin method, which has the following signature, returns the application skin set for the application or null if there is no JadeSkinApplication object set.

```java
getApplicationSkin(): JadeSkinApplication;
```

**Note** The Application::getSkin method returns a JadeSkin object. Call the getSkin method only if the skin was set by using the Application::setSkin method available before JADE release 6.0.

Setting the Skin of a Form

You can use the Form class setFormSkin method to set the current skin for a specific form regardless of the setting of the Window::skinCategoryName property or the Form class setApplicationSkin method to set the skin for a specific form and its controls. For details, see the following subsections.

**Form::setFormSkin Method**

The Form class setFormSkin method, which has the following signature, sets the current skin for a specific form regardless of the setting of the Window::skinCategoryName property.

```java
setFormSkin(skin: JadeSkinForm);
```

However, if the value of the Window class ignoreSkin property is set to true, the skin is still ignored. The skin has no impact on the controls of the form except potentially for the JadeSkinArea class backColor property (for more information, see "JadeSkinControl Class", earlier in this chapter).

The controls continue to use the any application-defined skins. Setting a specific skin for a form takes precedence over any defined application skin for that form.

**Note** Changing a skin object after the setFormSkin method is called has no impact on the displayed skin. To apply any skin changes dynamically, you must call the Form class setFormSkin method again.

To clear the form skin (cancel the skin display), call this method again with a null value, as follows.

```java
Form.setFormSkin(null);
```

The form then reverts to the use of an appropriate form skin set by the application.

**Form::setApplicationSkin Method**

The Form class setApplicationSkin method, which has the following signature, sets the skin for a specific form and its controls.

```java
setApplicationSkin(skinapp: JadeSkinApplication);
```

When you call this method to set a skin for a form and all of its controls, the skins used to draw the form and its controls are from the application skin specified in the skinapp parameter and any skin set by calling the Application class setApplicationSkin is ignored for this form and its controls.

If the application skin of the form does not include an appropriate skin for a control type, that control is not drawn with a skin.

**Note** Any control that has had a specific skin set by calling the Control class setSkin method continues to use that specific skin. The form application skin is used only if the setSkin(null) instruction is subsequently called on that control.
Setting the Skin of a Control

The Control class setSkin method, which has the following signature, sets the current skin for a specific control regardless of the setting of the Window:skinCategoryName property and the applyCondition criteria.

```java
setSkin(skin: JadeSkinControl);
```

However, if the value of the Window class ignoreSkin property is set to true, the skin is still ignored. Setting a specific skin for a control takes precedence over any defined application skin. Note that the skin object passed during this method call must correspond to the control type. For example, it must be of type JadeSkinFrame when the control is a Frame control class.

**Note** Changing a skin object after the setSkin method is called has no impact on the displayed skin. To apply any skin changes dynamically, you must call the Control class setSkin method again.

To clear the control skin (cancel the skin display), call this method again with a null value, as follows.

```java
Control.setSkin(null);
```

The control then reverts to the use of an appropriate control skin set for the application.

Ignoring a Skin

The Window class ignoreSkin property determines whether the window uses a skin. By default, this property is set to false and the window (form or control) uses any appropriate skin.

When you set the ignoreSkin property to true, the window is drawn without a skin.

Extracting and Loading Skins

The Extract Skins check box on the Schema Options sheet of the Extract dialog extracts all skins defined in your database. For details about extracting JADE skins for deployed runtime applications from your JADE development environment, see "Specifying Your Schema Options" under "Extracting Your Schema", in Chapter 10 of the JADE Development Environment User’s Guide.

If you want to load skins into an environment only for the initial load or when there have been many changes made to a skin, see "Using the JADE Skin Loader", in the following subsection.

If you want to load skins individually, use the Jade Skin Maintenance dialog. (For details, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the JADE Runtime Application Guide.)

Using the JADE Skin Loader

The schema load operation replaces existing skin entities of the same name.

If you want to load skins into an environment only for the initial load or when there have been many changes made to a skin, use the JADE Skin Loader. (The Jade Skin Maintenance dialog enables you to load skins individually. For details, see "Defining and Maintaining JADE Skins at Run Time", in Chapter 2 of the JADE Runtime Application Guide.)

The Application class skinLoad method displays the common Browse for Folder dialog, which prompts you to specify the root directory in which the skin images are located.

The name of the root directory is used as the application name for the skin. Although you can load a partial set of skins, you must load the complete set of skins for a specific control, menu item, or form because any existing skins with the same name are deleted prior to loading in the new skins.
**Note** The skins are not loaded unless the skins directory has the specified structure. For details, see "Directory Structure Example for the Button Control" or "Directory Structure Example for the Combo Box Control", in the following subsections.

For details about creating an empty skin directory structure into which you can load your skin image files of a specified name in a selected root directory, see the `Application` class `skinMakeDirectory` method.

At the end of the load process, a validation phase logs details about the load. The file name is the skin name with a `.log` file suffix (for example, `DemoSkin.log`), and it is located in the root directory in which the skin images are located (for example, the `DemoSkin` directory). You should review this file, because it lists skins that have not been loaded into the system.

**Note** The `skinLoad` method loads images only. You must change other settings by using the Jade Skin Maintenance dialog after the load is complete.

The code shown in the following fragment invokes the `skinLoad` method from a Workspace.

```java
app.skinLoad;
```

The following image shows the Jade Skin Maintenance dialog after the loading the Windows Broadbean skin.

The `FormAdminMdi` class `zSetupSkinSelectMenu` and `mnuSkin_click` methods in the `ErewhonInvestmentsViewSchema` schema, included in the `erewhon` subdirectory of the `examples` directory on the JADE release medium, provide an example of adding a skin for selection by users in runtime applications.
Directory Structure Example for the Button Control

The entries enclosed within the less than (<) and greater than (>) symbols in the following directory structure example indicate directories.

```
<SkinName>
<Button>  JadeSkinButton(<SkinName>_BN)
  imgBorderBottomLeft.png
  imgBorderBottomRight.png
  imgBorderBottomStrip.png
  imgBorderLeftStrip.png
  imgBorderRightStrip.png
  imgBorderTopLeft.png
  imgBorderTopRight.png
  imgBorderTopStrip.png
  imgInner.png
</Button>
  <up>  JadeSkinWindowStateImage(<SkinName>_BN_UP)
  imgBorderBottomLeft.png
  imgBorderBottomRight.png
  imgBorderBottomStrip.png
  imgBorderLeftStrip.png
  imgBorderRightStrip.png
  imgBorderTopLeft.png
  imgBorderTopRight.png
  imgBorderTopStrip.png
  imgInner.png
  imgPictureClosed.png
  imgPictureLeaf.png
  imgPictureMinus.png
  imgPictureOpen.png
  imgPicturePlus.png
</up>
  <down>  JadeSkinWindowStateImage(<SkinName>_BN_DN)
  imgBorderBottomLeft.png
  imgBorderBottomRight.png
  imgBorderBottomStrip.png
  imgBorderLeftStrip.png
  imgBorderRightStrip.png
  imgBorderTopLeft.png
  imgBorderTopRight.png
  imgBorderTopStrip.png
  imgInner.png
  imgPictureClosed.png
  imgPictureLeaf.png
  imgPictureMinus.png
  imgPictureOpen.png
  imgPicturePlus.png
...  // and so on
```

For details about creating an empty skin directory structure into which you can load your skin image files, see the Application class skinMakeDirectory method, in Chapter 1 of the JADE Encyclopaedia of Classes.
Directory Structure Example for the Combo Box Control

The entries enclosed within the less than (<) and greater than (>) symbols in the following directory structure example indicate directories.

```xml
<SkinName>
  <ComboBox>← JadeSkinComboBox (<SkinName>_CB)
    imgBorderBottomLeft.png
    imgBorderBottomRight.png
    imgBorderBottomStrip.png
    imgBorderLeftStrip.png
    imgBorderRightStrip.png
    imgBorderTopLeft.png
    imgBorderTopRight.png
    imgBorderTopStrip.png
    imgInner.png
  </ComboBox>
  <SimpleButton>← JadeSkinSimpleButton (<SkinName>_CB_BN)
    imgDisabled.png
    imgDown.png
    imgRollOver.png
    imgUp.png
  </SimpleButton>
  <ListBox>← JadeSkinListBox (<SkinName>_CB_LB)
    imgBorderBottomLeft.png
    imgBorderBottomRight.png
    imgBorderBottomStrip.png
    imgBorderLeftStrip.png
    imgBorderRightStrip.png
    imgBorderTopLeft.png
    imgBorderTopRight.png
    imgBorderTopStrip.png
    imgInner.png
    imgPictureClosed.png
    imgPictureLeaf.png
    imgPictureMinus.png
    imgPictureOpen.png
    imgPicturePlus.png
  </ListBox>
  <TextBox>← JadeSkinTextBox (<SkinName>_CB_TB)
    imgBorderBottomLeft.png
    imgBorderBottomRight.png
    imgBorderBottomStrip.png
    imgBorderLeftStrip.png
    imgBorderRightStrip.png
    imgBorderTopLeft.png
    imgBorderTopRight.png
    imgBorderTopStrip.png
    imgInner.png
    imgPictureClosed.png
    imgPictureLeaf.png
    imgPictureMinus.png
    imgPictureOpen.png
    imgPicturePlus.png
</SkinName>
```
Naming Convention when Loading JADE Skins

The following table lists the conventions that JADE uses to name `JadeSkinEntity` objects. For example, the `JadeSkinComboBox` subclass of the `JadeSkinControl` class has four separate objects created for it, each one with the following format:

```
application-skin-name_acronym
```

For example, a border object for the `JadeSkinButton` object in the application skin `DemoSkin` is appended with the `BN` acronym (that is, `DemoSkin_BN`) and the `JadeSkinWindowStateImage` of the `myButtonUp` property of that object is further appended by `UP` (that is, `DemoSkin_BN_UP`).

<table>
<thead>
<tr>
<th>JadeSkinEntity</th>
<th>Images</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>JadeSkinBaseControl</td>
<td>Border (nine images)</td>
<td>BC</td>
</tr>
<tr>
<td>JadeSkinBrowseButtons</td>
<td>Border (nine images)</td>
<td>BB</td>
</tr>
<tr>
<td></td>
<td>First (JadeSkinSimpleButton)</td>
<td>BB_FI</td>
</tr>
<tr>
<td></td>
<td>Prior (JadeSkinSimpleButton)</td>
<td>BB_PR</td>
</tr>
<tr>
<td></td>
<td>Next (JadeSkinSimpleButton)</td>
<td>BB_NX</td>
</tr>
<tr>
<td></td>
<td>Last (JadeSkinSimpleButton)</td>
<td>BB_LA</td>
</tr>
<tr>
<td>JadeSkinButton</td>
<td>Border (nine images)</td>
<td>BN</td>
</tr>
<tr>
<td></td>
<td>Up (JadeSkinWindowStateImage)</td>
<td>BN_UP</td>
</tr>
<tr>
<td></td>
<td>Disabled (JadeSkinWindowStateImage)</td>
<td>BN_DI</td>
</tr>
<tr>
<td></td>
<td>Down (JadeSkinWindowStateImage)</td>
<td>BN_DO</td>
</tr>
<tr>
<td></td>
<td>Focus (JadeSkinWindowStateImage)</td>
<td>BN_FO</td>
</tr>
<tr>
<td></td>
<td>Focus down (JadeSkinWindowStateImage)</td>
<td>BN_FD</td>
</tr>
<tr>
<td></td>
<td>Roll over (JadeSkinWindowStateImage)</td>
<td>BN_RO</td>
</tr>
<tr>
<td></td>
<td>Roll under (JadeSkinWindowStateImage)</td>
<td>BN_RU</td>
</tr>
<tr>
<td>JadeSkinCheckBox</td>
<td>Border (nine images)</td>
<td>CH</td>
</tr>
<tr>
<td></td>
<td>True (JadeSkinSimpleButton)</td>
<td>CH_TR</td>
</tr>
<tr>
<td></td>
<td>False (JadeSkinSimpleButton)</td>
<td>CH_FA</td>
</tr>
<tr>
<td>JadeSkinComboBox</td>
<td>Border (nine images)</td>
<td>CB</td>
</tr>
<tr>
<td></td>
<td>Button (JadeSkinSimpleButton)</td>
<td>CB_BN</td>
</tr>
<tr>
<td></td>
<td>List box (JadeSkinListBox)</td>
<td>CB_LB</td>
</tr>
<tr>
<td></td>
<td>Text box (JadeSkinTextBox)</td>
<td>CB_TB</td>
</tr>
<tr>
<td>JadeSkinFolder</td>
<td>Border (nine images)</td>
<td>FO</td>
</tr>
</tbody>
</table>
## JadeSkinEntity

<table>
<thead>
<tr>
<th>JadeSkinEntity</th>
<th>Images</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>JadeSkinButton</td>
<td>Tab button (JadeSkinButton)</td>
<td>FO_BN</td>
</tr>
<tr>
<td>JadeSkinFrame</td>
<td>Border (nine images)</td>
<td>FR</td>
</tr>
<tr>
<td>JadeSkinGroupBox</td>
<td>Border (nine images)</td>
<td>GB</td>
</tr>
<tr>
<td></td>
<td>Caption (JadeSkinLabel)</td>
<td>GB_LA</td>
</tr>
<tr>
<td>JadeSkinHScrollBar</td>
<td>Border (nine images)</td>
<td>HS</td>
</tr>
<tr>
<td></td>
<td>Left button (JadeSkinSimpleButton)</td>
<td>HS_LE</td>
</tr>
<tr>
<td></td>
<td>Right button (JadeSkinSimpleButton)</td>
<td>HS_RI</td>
</tr>
<tr>
<td></td>
<td>Thumb track up (JadeSkinWindowStateImage)</td>
<td>HS_UP</td>
</tr>
<tr>
<td></td>
<td>Thumb track disabled (JadeSkinWindowStateImage)</td>
<td>HS_DI</td>
</tr>
<tr>
<td></td>
<td>Thumb track down (JadeSkinWindowStateImage)</td>
<td>HS_DO</td>
</tr>
<tr>
<td></td>
<td>Thumb track rollover (JadeSkinWindowStateImage)</td>
<td>HS_RO</td>
</tr>
<tr>
<td>JadeSkinDockBar</td>
<td>Border (nine images)</td>
<td>JB</td>
</tr>
<tr>
<td></td>
<td>Horizontal grip bar (JadeSkinWindowStateImage)</td>
<td>JB_HG</td>
</tr>
<tr>
<td></td>
<td>Horizontal resize bar (JadeSkinWindowStateImage)</td>
<td>JB_HB</td>
</tr>
<tr>
<td></td>
<td>Vertical grip bar (JadeSkinWindowStateImage)</td>
<td>JB_VG</td>
</tr>
<tr>
<td></td>
<td>Vertical resize bar (JadeSkinWindowStateImage)</td>
<td>JB_VB</td>
</tr>
<tr>
<td>JadeSkinDockContainer</td>
<td>Border (nine images)</td>
<td>JC</td>
</tr>
<tr>
<td></td>
<td>Horizontal grip bar (JadeSkinWindowStateImage)</td>
<td>JC_HG</td>
</tr>
<tr>
<td></td>
<td>Horizontal resize bar (JadeSkinWindowStateImage)</td>
<td>JC_HB</td>
</tr>
<tr>
<td></td>
<td>Vertical grip bar (JadeSkinWindowStateImage)</td>
<td>JC_VG</td>
</tr>
<tr>
<td></td>
<td>Vertical resize bar (JadeSkinWindowStateImage)</td>
<td>JC_VB</td>
</tr>
<tr>
<td>JadeSkinEditMask</td>
<td>Border (nine images)</td>
<td>JE</td>
</tr>
<tr>
<td>JadeSkinMask</td>
<td>Button (JadeSkinButton)</td>
<td>JM</td>
</tr>
<tr>
<td>JadeSkinRichText</td>
<td>Border (nine images)</td>
<td>JR</td>
</tr>
<tr>
<td>JadeSkinLabel</td>
<td>Border (nine images)</td>
<td>LA</td>
</tr>
</tbody>
</table>
### JadeSkinEntity

<table>
<thead>
<tr>
<th>JadeSkinEntity</th>
<th>Images</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>JadeSkinListBox</td>
<td>Border (nine images)</td>
<td>LB</td>
</tr>
<tr>
<td></td>
<td>Picture (five images)</td>
<td>LB_PI</td>
</tr>
<tr>
<td>JadeSkinOleControl</td>
<td>Border (nine images)</td>
<td>OC</td>
</tr>
<tr>
<td>JadeSkinOptionButton</td>
<td>Border (nine images)</td>
<td>OB</td>
</tr>
<tr>
<td></td>
<td>True (JadeSkinSimpleButton)</td>
<td>OB_TR</td>
</tr>
<tr>
<td></td>
<td>False (JadeSkinSimpleButton)</td>
<td>OB_FA</td>
</tr>
<tr>
<td>JadeSkinPicture</td>
<td>Border (nine images)</td>
<td>PI</td>
</tr>
<tr>
<td>JadeSkinProgressBar</td>
<td>Border (nine images)</td>
<td>BN</td>
</tr>
<tr>
<td></td>
<td>Progress (JadeSkinWindowStateImage)</td>
<td>PB_PR</td>
</tr>
<tr>
<td>JadeSkinSheet</td>
<td>Border (nine images)</td>
<td>SH</td>
</tr>
<tr>
<td></td>
<td>Tab button (JadeSkinButton)</td>
<td>SH_BN</td>
</tr>
<tr>
<td>JadeSkinStatusLine</td>
<td>Border (nine images)</td>
<td>SL</td>
</tr>
<tr>
<td>JadeSkinTable</td>
<td>Border (nine images)</td>
<td>TA</td>
</tr>
<tr>
<td>JadeSkinTextBox</td>
<td>Border (nine images)</td>
<td>TB</td>
</tr>
<tr>
<td>JadeSkinVScroll</td>
<td>Border (nine images)</td>
<td>VS</td>
</tr>
<tr>
<td></td>
<td>Top button (JadeSkinSimpleButton)</td>
<td>VS_TO</td>
</tr>
<tr>
<td></td>
<td>Bottom button (JadeSkinSimpleButton)</td>
<td>VS_BO</td>
</tr>
<tr>
<td></td>
<td>Thumb track up (JadeSkinWindowStateImage)</td>
<td>VS_UP</td>
</tr>
<tr>
<td></td>
<td>Thumb track disabled (JadeSkinWindowStateImage)</td>
<td>VS_DI</td>
</tr>
<tr>
<td></td>
<td>Thumb track down (JadeSkinWindowStateImage)</td>
<td>VS_DO</td>
</tr>
<tr>
<td></td>
<td>Thumb track rollover (JadeSkinWindowStateImage)</td>
<td>VS_RO</td>
</tr>
<tr>
<td>JadeSkinMenu</td>
<td>Border (nine images)</td>
<td>MN</td>
</tr>
<tr>
<td></td>
<td>Check mark image</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Separator image</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Submenu arrow image</td>
<td></td>
</tr>
</tbody>
</table>
### JadeSkinEntity

<table>
<thead>
<tr>
<th>Entity</th>
<th>Images</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>JadeSkinForm</td>
<td>Active border (11 images)</td>
<td>FM</td>
</tr>
<tr>
<td></td>
<td>Inactive border (eight images)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inner image</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximize (JadeSkinSimpleButton)</td>
<td>FM_MX</td>
</tr>
<tr>
<td></td>
<td>Maximized (JadeSkinSimpleButton)</td>
<td>FM_MD</td>
</tr>
<tr>
<td></td>
<td>Minimize (JadeSkinSimpleButton)</td>
<td>FM_MI</td>
</tr>
<tr>
<td></td>
<td>Terminate (JadeSkinSimpleButton)</td>
<td>FM_TE</td>
</tr>
<tr>
<td></td>
<td>Popup menu (JadeSkinMenu)</td>
<td>FM_PM</td>
</tr>
<tr>
<td></td>
<td>MDI minimize (JadeSkinSimpleButton)</td>
<td>FM_MN</td>
</tr>
<tr>
<td></td>
<td>MDI restore (JadeSkinSimpleButton)</td>
<td>FM_RE</td>
</tr>
<tr>
<td></td>
<td>MDI terminate (JadeSkinSimpleButton)</td>
<td>FM_MT</td>
</tr>
<tr>
<td>JadeSkinWindowStateImage</td>
<td>Border (8 images)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Inner picture image</td>
<td></td>
</tr>
<tr>
<td>JadeSkinButton</td>
<td>Up, disabled, down, and over images</td>
<td></td>
</tr>
</tbody>
</table>

All entities other than **JadeSkinMenu** also have an image mask image.

## Assigning Categories

You can embed category names within the directory name by appending the category name to the control name.

In the following examples, **DemoSkin** is the application skin name and **Plain** and **Flowery** are category names.

- **DemoSkin_Plain**
- **DemoSkin_Flowery**

The total length of the skin name and category name cannot exceed 20 characters.

The skin name must be unique across the JADE system so that it is the application skin name, optionally appended by the category name, and followed by sufficient acronyms to make it unique. For example, a combo box that is part of the **DemoSkin** skin and has a category of **Flowery** has the following generated name.

- **DemoSkin_Flowery_CB**

The associated text box has the following name.

- **DemoSkin_Flowery_CB_TB**

Both entities are assigned to the **Flowery** category.
Chapter 10  Synchronized Database Service (SDS) Development Considerations

This chapter covers the following topics.

- **Overview**
- **Using the JadeDatabaseAdmin Class SDS Methods**
  - JadeDatabaseAdmin Methods
  - Returning Information about SDS Entities
    - SDSPrimary Dynamic Object Type
    - SDSSecondary Dynamic Object Type
    - SDSSecondaryProxy Dynamic Object Type
    - SDSTransaction Dynamic Object Type
    - SDS Global Constants
- **Developing Applications to Operate in an SDE**
  - Inter-System Event Notifications
    - Persistent Events and Notifications
    - Communicating from a Secondary to a Primary System
  - Distinguishing Shadow Transactions from Reader Processes
  - Detecting SDS Role Changes
  - Journal Transfer Halted Events
  - Connection State Change Events
  - Tracking Change Events
  - Inquiry Access to Secondary Database Servers
  - Automatically Starting Server Applications in Secondary Systems
  - Stopping and Starting Server Applications in Secondary Systems

**Overview**

You can use the SDS methods provided by the [JadeDatabaseAdmin](#) class to integrate Synchronized Database Service (SDS) monitoring and administrative services into your own applications or to build a standalone application to administer a Synchronized Database Environment (SDE).

**Caution**  SDS is limited to keeping the database (that is, the *.dat) files synchronized.

It is your responsibility to keep any other files on which your applications rely synchronized; for example, JADE binaries, user DLLs, ActiveX controls, text files, pictures, documents, and so on.
You can call the JadeDatabaseAdmin class methods used to monitor and control SDS operations in an SDS-enabled system from primary and secondary systems, but some methods apply only to one database role or the other. For details about the Synchronized Database Service, see Chapter 1, "Administering a JADE Synchronized Database Service (SDS) Environment", in the JADE Synchronized Database Service (SDS) Administration Guide.

Using the JadeDatabaseAdmin Class SDS Methods

This section contains the following topics.

- JadeDatabaseAdmin Methods
- Returning Information about SDS Entities
  - SDSPrimary Dynamic Object Type
  - SDSSecondary Dynamic Object Type
  - SDSSecondaryProxy Dynamic Object Type
  - SDSTransaction Dynamic Object Type
- SDS Global Constants
  - JadeDynamicObjectNames Category Global Constants
  - JadeDynamicObjectTypes Category Global Constants
  - JadeErrorCodesSDS Category Global Constants
  - SDSConnectionState
  - SDSDatabaseRoles
  - SDSEventTypes
  - SDSReorgState
  - SDSSecondaryState
  - SDSStopTrackingCodes
  - SDSTakeoverState
  - SDSTransactionStates

For details, see the following subsections.

JadeDatabaseAdmin Methods

The JadeDatabaseAdmin class provides the methods summarized in the following table, which you can call to monitor and control SDS operations in an SDS-enabled system.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>Valid at …</th>
</tr>
</thead>
<tbody>
<tr>
<td>getReasonTrackingStoppedString</td>
<td>Returns a string containing a textual description of the SDSStopTrackingCodes global constant reason code</td>
<td>Anywhere</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
<td>Valid at …</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>sdsAuditStopTracking</td>
<td>Specifies a number that is returned in the <code>userInfo</code> parameter of your user notification method when an <code>SDS_TrackingStopped</code> event occurs</td>
<td>Anywhere</td>
</tr>
<tr>
<td>sdsDisablePrimaryConnection</td>
<td>Disables a connection from the current secondary database to the primary server</td>
<td>Secondary only</td>
</tr>
<tr>
<td>sdsDisablePrimaryConnectionAt</td>
<td>Disables a connection from a specified secondary database to the primary server</td>
<td>Primary only</td>
</tr>
<tr>
<td>sdsDisableReadAccess</td>
<td>Disallows read-only database access to this secondary database</td>
<td>Secondary only</td>
</tr>
<tr>
<td>sdsDisableReadAccessAt</td>
<td>Disallows read-only database access at a specified secondary database</td>
<td>Primary only</td>
</tr>
<tr>
<td>sdsEnableReadAccess</td>
<td>Allows read-only database access to this secondary database</td>
<td>Secondary only</td>
</tr>
<tr>
<td>sdsEnableReadAccessAt</td>
<td>Allows read-only database access at a specified secondary database</td>
<td>Primary only</td>
</tr>
<tr>
<td>sdsGetDatabaseRole</td>
<td>Returns the database role of the current server for the JADE system</td>
<td>Anywhere</td>
</tr>
<tr>
<td>sdsGetDatabaseSubrole</td>
<td>Returns the database subrole of the current server for the JADE system</td>
<td>Anywhere</td>
</tr>
<tr>
<td>sdsGetMyServerInfo</td>
<td>Obtains an array describing the SDS attributes of the JADE system</td>
<td>Anywhere</td>
</tr>
<tr>
<td>sdsGetSecondaryInfo</td>
<td>Obtains an array containing the SDS attributes for a specified secondary system</td>
<td>Primary only</td>
</tr>
<tr>
<td>sdsGetSecondaryProxies</td>
<td>Obtains an array of secondary proxy dynamic objects</td>
<td>Primary only</td>
</tr>
<tr>
<td>sdsGetSecondaryProxy</td>
<td>Obtains information about a specific secondary proxy dynamic object</td>
<td>Primary only</td>
</tr>
<tr>
<td>sdsGetTransactions</td>
<td>Obtains an array of transaction dynamic objects on the current secondary system</td>
<td>Secondary only</td>
</tr>
<tr>
<td>sdsGetTransactionsAt</td>
<td>Obtains an array of transaction dynamic objects on a specified secondary system</td>
<td>Primary only</td>
</tr>
<tr>
<td>sdsInitiateHostileTakeover</td>
<td>Initiates a hostile take-over by the executing secondary system</td>
<td>Secondary only</td>
</tr>
<tr>
<td>sdsInitiateTakeover</td>
<td>Initiates a negotiated take-over by a specified secondary system so that server becomes the primary system</td>
<td>Primary only</td>
</tr>
<tr>
<td>sdsIsInitialized</td>
<td>Returns <code>true</code> if SDS is initialized for the system</td>
<td>Anywhere</td>
</tr>
<tr>
<td>sdsIsRunning</td>
<td>Returns <code>true</code> if SDS is running for this system</td>
<td>Anywhere</td>
</tr>
<tr>
<td>sdsReconnectNow</td>
<td>Prompts the secondary database to attempt a reconnect to its primary server</td>
<td>Secondary only</td>
</tr>
<tr>
<td>sdsReplayNextJournal</td>
<td>Initiates a replay of the next ready journal on a secondary database when journal replay is suspended</td>
<td>Secondary only</td>
</tr>
</tbody>
</table>
## Returning Information about SDS Entities

SDS node information is returned in instances of `JadeDynamicObject`. A JADE dynamic object is a self-describing object whose attributes can be specified at run time.

The runtime type of a `JadeDynamicObject` is specified by the `type` attribute and the `name` attribute contains the name of the object in textual format. For details, see the `JadeDynamicObject` and `JadeDatabaseAdmin` classes, in Chapter 1 of the *JADE Encyclopaedia of Classes*.

The following dynamic object types are returned from `JadeDatabaseAdmin` class SDS query methods.

- `SDSPrimary`
- `SDSSecondary`
- `SDSSecondaryProxy`
- `SDSTransaction`

For details, see the following subsections.

### SDSPrimary Dynamic Object Type

The `SDSPrimary` dynamic object type has a `name` attribute value of `SDSPrimary` and a `type` attribute value of `SDS_PrimaryType (1)`. The dynamic attributes for this primary object type are listed in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connectedSecondaryServers</td>
<td>Integer</td>
<td>Number of secondary servers connected to the primary</td>
</tr>
<tr>
<td>currentJournalNumber</td>
<td>Integer</td>
<td>Number of the current write journal</td>
</tr>
</tbody>
</table>
Chapter 10  Synchronized Database Service (SDS) Development Considerations

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentJournalTimeStamp</td>
<td>TimeStamp</td>
<td>Timestamp of the current journal converted to a local value</td>
</tr>
<tr>
<td>currentJournalTimeStampUTC</td>
<td>TimeStamp</td>
<td>Timestamp of the current journal as a UTC value</td>
</tr>
<tr>
<td>latestCommittedTimestamp</td>
<td>TimeStamp</td>
<td>Timestamp of the latest commit audit record appended to the audit trail converted to a local value</td>
</tr>
<tr>
<td>latestCommittedTimestampUTC</td>
<td>TimeStamp</td>
<td>Timestamp of the latest commit audit record appended to the audit trail as a UTC value</td>
</tr>
<tr>
<td>maxCommittedTranID</td>
<td>Decimal</td>
<td>Highest transaction id committed by the primary database, which may not be the latest transaction committed</td>
</tr>
<tr>
<td>myHostName</td>
<td>String</td>
<td>Computer name of the primary host</td>
</tr>
<tr>
<td>myName</td>
<td>String</td>
<td>Name of the primary, specified in the MyName parameter in the [SyncDbService] section of the JADE initialization file</td>
</tr>
</tbody>
</table>

The audit timestamp attributes with a UTC suffix hold UTC values that are not converted to local time. The audit timestamp attributes without a UTC suffix hold UTC values that are converted to local time. The timestamps of the primary and secondary are derived from the UTC audit timestamp recorded by the primary and converted to local time on the secondary (catering for primary and secondary time zones that differ).

**SDSSecondary Dynamic Object Type**

The SDSSecondary dynamic object type has a name attribute value of SDSSecondary and a type attribute value of SDS_Secondary (3). The SDSSecondary dynamic object returned by the sdsGetSecondaryInfo method called on a primary or the sdsGetMyServerInfo method called on a secondary is always populated on the secondary, and it therefore has the same attributes.

The dynamic attributes for this secondary object type for both the block write and journal switch synchronization modes are listed in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>activeTransactions</td>
<td>Integer</td>
<td>Number of active transactions</td>
</tr>
<tr>
<td>connectionCheckInterval</td>
<td>Integer</td>
<td>Number of seconds at which the secondary database polls the primary to determine reachability via the communication paths, specified in the ConnectionPollInterval parameter in the [SyncDbService] section of the JADE initialization file</td>
</tr>
<tr>
<td>connectionState</td>
<td>Integer</td>
<td>State of the connection to the primary (see the first of the following tables)</td>
</tr>
<tr>
<td>currentReplayJournalNumber</td>
<td>Integer</td>
<td>Number of the journal currently replaying</td>
</tr>
<tr>
<td>currentReplayJournalTimeStamp</td>
<td>TimeStamp</td>
<td>Timestamp of the journal currently replaying converted to a local value</td>
</tr>
<tr>
<td>currentReplayJournalTimeStampUTC</td>
<td>TimeStamp</td>
<td>Timestamp of the journal currently replaying as a UTC value</td>
</tr>
<tr>
<td>interruptedTransactions</td>
<td>Integer</td>
<td>Number of interrupted transactions</td>
</tr>
<tr>
<td>lastErrorCode</td>
<td>Integer</td>
<td>Number of the last error that occurred</td>
</tr>
<tr>
<td>lastReplayJournalNumber</td>
<td>Integer</td>
<td>Number of the last journal that was replayed</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>lastReplayJournalTimeStamp</td>
<td>TimeStamp</td>
<td>Timestamp of the last journal that was replayed converted to a local value</td>
</tr>
<tr>
<td>lastReplayJournalTimeStampUTC</td>
<td>TimeStamp</td>
<td>Timestamp of the last journal that was replayed as a UTC value</td>
</tr>
<tr>
<td>latestReadyJournalNumber</td>
<td>Integer</td>
<td>Number of the latest journal that is ready to be replayed</td>
</tr>
<tr>
<td>latestReadyJournalTimeStamp</td>
<td>TimeStamp</td>
<td>Timestamp of the latest journal that is ready to be replayed converted to a local value</td>
</tr>
<tr>
<td>latestReadyJournalTimeStampUTC</td>
<td>TimeStamp</td>
<td>Timestamp of the latest journal that is ready to be replayed as a UTC value</td>
</tr>
<tr>
<td>latestReplayedAuditTimeStamp</td>
<td>TimeStamp</td>
<td>Timestamp of the last audit record replayed by the database tracker converted to a local value</td>
</tr>
<tr>
<td>latestReplayedAuditTimeStampUTC</td>
<td>TimeStamp</td>
<td>Timestamp of the last audit record replayed by the database tracker as a UTC value</td>
</tr>
<tr>
<td>latestStableAuditTimeStamp</td>
<td>TimeStamp</td>
<td>Timestamp of the latest audit record written to disk in block write mode converted to a local value</td>
</tr>
<tr>
<td>latestStableAuditTimeStampUTC</td>
<td>TimeStamp</td>
<td>Timestamp of the latest audit record written to disk in block write mode as a UTC value</td>
</tr>
<tr>
<td>myHostName</td>
<td>String</td>
<td>Computer name of the secondary host</td>
</tr>
<tr>
<td>myName</td>
<td>String</td>
<td>Name of the secondary, specified in the MyName parameter in the [SyncDbService] section of the JADE initialization file</td>
</tr>
<tr>
<td>nextReplayJournalNumber</td>
<td>Integer</td>
<td>Number of the next journal to replay</td>
</tr>
<tr>
<td>nextReplayJournalTimeStamp</td>
<td>TimeStamp</td>
<td>Timestamp of the next journal to replay converted to a local value</td>
</tr>
<tr>
<td>nextReplayJournalTimeStampUTC</td>
<td>TimeStamp</td>
<td>Timestamp of the next journal to replay as a UTC value</td>
</tr>
<tr>
<td>primaryHostName</td>
<td>String</td>
<td>Computer name of the primary host</td>
</tr>
<tr>
<td>primaryServerName</td>
<td>String</td>
<td>Name of the primary, specified in the PrimaryServerName parameter in the [SyncDbService] section of the JADE initialization file</td>
</tr>
<tr>
<td>readAccessDisabled</td>
<td>Boolean</td>
<td>Specifies whether read access is disabled (the value of the ReadAccessDisabled parameter in the [SyncDbService] section of the JADE initialization file)</td>
</tr>
<tr>
<td>readAccessGranted</td>
<td>Boolean</td>
<td>Specifies whether read access has been granted</td>
</tr>
<tr>
<td>reasonTrackingStopped</td>
<td>Integer</td>
<td>Reason tracking stopped (see the second of the following tables)</td>
</tr>
<tr>
<td>reconnectInterval</td>
<td>Integer</td>
<td>Frequency (in seconds) at which a secondary database server attempts to reconnect to its primary server when a primary server is not available (the value of the ReconnectInterval parameter in the [SyncDbService] section of the JADE initialization file)</td>
</tr>
<tr>
<td>recoveryRequired</td>
<td>Boolean</td>
<td>Specifies whether recovery is required</td>
</tr>
<tr>
<td>Name</td>
<td>Type</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>----------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>reorgStatus</td>
<td>Integer</td>
<td>Reorganization status (see the third of the following tables)</td>
</tr>
<tr>
<td>rpsStorageMode</td>
<td>Integer</td>
<td>Storage mode represented by one of the RpsStorageMode_Full (0), RpsStorageMode_MappedExtent (1), or RpsStorageMode_WorkingSet (2)</td>
</tr>
<tr>
<td>rpsTransitionHaltCode</td>
<td>Integer</td>
<td>RPS transition halt code (see the fourth of the following tables)</td>
</tr>
<tr>
<td>rpsWorkers</td>
<td>Integer</td>
<td>Number of RPS worker threads</td>
</tr>
<tr>
<td>state</td>
<td>Integer</td>
<td>State of the secondary in relation to the primary (see the fifth of the following tables)</td>
</tr>
<tr>
<td>subrole</td>
<td>Integer</td>
<td>Database role (see the sixth of the following tables)</td>
</tr>
<tr>
<td>syncMode</td>
<td>Integer</td>
<td>Mode of journal synchronization, specified in the SyncMode parameter in the [SyncDbService] section of the JADE initialization file</td>
</tr>
<tr>
<td>tracking</td>
<td>Boolean</td>
<td>Contains true when tracking is active or it contains false if tracking is stopped for any reason</td>
</tr>
<tr>
<td>trackingDisabled</td>
<td>Boolean</td>
<td>Specifies whether database tracking (journal replay) process is disabled (the value of the TrackingDisabled parameter in the [SyncDbService] section of the JADE initialization file)</td>
</tr>
</tbody>
</table>

The values of the connectionState attribute are represented by one of the SDSConnectionState category global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_Connected</td>
<td>2</td>
</tr>
<tr>
<td>SDS_Connecting</td>
<td>3</td>
</tr>
<tr>
<td>SDS_ConnectionFailed</td>
<td>4</td>
</tr>
<tr>
<td>SDS_ConnectionStateUndefined</td>
<td>0</td>
</tr>
<tr>
<td>SDS_Disconnected</td>
<td>1</td>
</tr>
</tbody>
</table>

The values of the reasonTrackingStopped attribute are represented by one of the SDSStopTrackingCodes category global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_ReasonAdminAudited</td>
<td>1</td>
<td>Tracking stopped because the database entered delta mode.</td>
</tr>
<tr>
<td>SDS_ReasonAdminDirect</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonAutoUpgradeMismatch</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonDeltaModeEntered</td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
The values of the `reorgStatus` attribute are represented by one of the `SDSReorgState` category global constants listed in the following table:

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_ReorgStateNotReorging</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>SDS_ReorgStateOfflinePhase</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>SDS_ReorgStateReorgingFiles</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SDS_ReorgStateRestarting</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>SDS_ReorgStateStarting</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SDS_ReorgStateSeekingApproval</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

The values of the `rpsTransitionHaltCode` attribute are represented by one of the `RPSTransitionHaltCode` category global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPS_HaltAutoScript</td>
<td>1</td>
<td>An automatic initiate alter script was generated (will be automatically loaded by the Datapump application if configured to automatically restart)</td>
</tr>
</tbody>
</table>
The values of the `state` attribute are represented by one of the `SDSSecondaryState` category global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPS_HaltManualScript</td>
<td>2</td>
<td>A manual alter script was generated (requires administration user intervention to apply changes to RDB before tracking can be resumed)</td>
</tr>
<tr>
<td>RPS_HaltMappingDeleted</td>
<td>3</td>
<td>The RPS mapping was deleted on the primary database, rendering the RPS node and associated RDB defunct</td>
</tr>
<tr>
<td>RPS_HaltNoScript</td>
<td>0</td>
<td>Changes do not affect RDB, so no script was generated</td>
</tr>
</tbody>
</table>

The values of the `subrole` attribute are represented by one of the `SDSDatabaseRoles` category global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_SubroleNative (native JADE Object Manager database)</td>
<td>1</td>
</tr>
<tr>
<td>SDS_SubroleRelational (relational database)</td>
<td>2</td>
</tr>
</tbody>
</table>

The values of the `syncMode` attribute are represented by one of the `SDSSecondaryState` category global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_BlockWrite</td>
<td>2</td>
</tr>
<tr>
<td>SDS_JournalSwitch</td>
<td>1</td>
</tr>
</tbody>
</table>

**Note** When a secondary server is restarted in an interrupted mode, the `recoveryRequired` attribute is set but the active and interrupted transaction counts are not valid until the first journal has been replayed. The `recoveryRequired` attribute is reset when outstanding interrupted transactions complete.

For the block write synchronization mode only, the dynamic attributes for the secondary object update type are listed in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>latestReplayedAuditTimestamp</td>
<td>timeStamp</td>
<td>Timestamp of the latest replayed audit record</td>
</tr>
<tr>
<td>latestStableAuditTimestamp</td>
<td>timeStamp</td>
<td>Timestamp of the latest stable replayed audit record</td>
</tr>
</tbody>
</table>
To obtain the latest committed timestamp for a secondary in block write synchronization mode, compare the `latestReplayedAuditTimestamp` attribute with the `latestStableAuditTimestamp` value, by a single call to the `JadeDatabaseAdmin` class `sdsGetSecondaryInfo` method from the primary or the `JadeDatabaseAdmin` class `sdsGetMyServerInfo` method from a secondary. The difference between these attribute values should be very small and under normal conditions, they will differ only by network latency plus the journal disk write time on the secondary.

All audit timestamps are UTC values, which are converted to local time for dynamic attributes. The timestamps of both the primary and secondary proxy are both derived from the UTC audit timestamp recorded by the primary and converted to local time on the secondary proxy (catering for primary and secondary proxy time zones that differ).

**SDSSecondaryProxy Dynamic Object Type**

The `SDSSecondaryProxy` dynamic object type has a name attribute value of `SDSSecondaryProxy` and a type attribute value of `SDS_SecondaryProxyType` (2).

The dynamic attributes for this secondary proxy object type are listed in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxCommittedTranID</td>
<td>Decimal</td>
<td>Transaction id of the maximum committed audit record</td>
</tr>
<tr>
<td>maxReplayedTranID</td>
<td>Decimal</td>
<td>Transaction id of the maximum replayed audit record</td>
</tr>
<tr>
<td>maxStableTranID</td>
<td>Decimal</td>
<td>Transaction id of the maximum stable audit record</td>
</tr>
<tr>
<td>connectionCheckInterval</td>
<td>Integer</td>
<td>Number of seconds at which the secondary database polls the primary to determine reachability via the communication paths, specified in the <code>ConnectionPollInterval</code> parameter in the [SyncDbService] section of the JADE initialization file</td>
</tr>
<tr>
<td>connectionState</td>
<td>Integer</td>
<td>State of the connection to the primary</td>
</tr>
<tr>
<td>hostName</td>
<td>String</td>
<td>Computer name of the secondary proxy host on the primary</td>
</tr>
<tr>
<td>lastErrorCode</td>
<td>Integer</td>
<td>Number of the last error that occurred</td>
</tr>
<tr>
<td>lastRecordSentJournal</td>
<td>Integer64</td>
<td>Journal number of the last journal block sent</td>
</tr>
<tr>
<td>lastRecordSentOffset</td>
<td>Integer64</td>
<td>Byte offset of the last journal block sent</td>
</tr>
<tr>
<td>myName</td>
<td>String</td>
<td>Name of the secondary proxy on the primary, specified in the <code>MyName</code> parameter in the [SyncDbService] section of the JADE initialization file</td>
</tr>
<tr>
<td>nextJournalNumber</td>
<td>Integer</td>
<td>Next journal the primary sends if the secondary is catching up or the next write journal when the secondary is mirroring writes from the current journal. It remains valid when the secondary is disconnected.</td>
</tr>
<tr>
<td>primaryServerName</td>
<td>String</td>
<td>Name of the primary, specified in the <code>PrimaryServerName</code> parameter in the [SyncDbService] section of the JADE initialization file</td>
</tr>
<tr>
<td>subrole</td>
<td>Integer</td>
<td>Database role</td>
</tr>
<tr>
<td>syncMode</td>
<td>Integer</td>
<td>Mode of journal synchronization, specified in the <code>SyncMode</code> parameter in the [SyncDbService] section of the JADE initialization file</td>
</tr>
<tr>
<td>totalSends</td>
<td>Integer64</td>
<td>Count of messages sent to the secondary</td>
</tr>
</tbody>
</table>
The values of the **subrole** attribute are represented by one of the **SDSDatabaseRoles** category global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_SubroleNative (native JADE Object Manager database)</td>
<td>1</td>
</tr>
<tr>
<td>SDS_SubroleRelational (relational database)</td>
<td>2</td>
</tr>
</tbody>
</table>

The values of the **syncMode** attribute are represented by one of the **SDSSecondaryState** category global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_BlockWrite</td>
<td>2</td>
</tr>
<tr>
<td>SDS_JournalSwitch</td>
<td>1</td>
</tr>
</tbody>
</table>

**SDSTransaction Dynamic Object Type**

The **SDSTransaction** dynamic object type has a **name** attribute value of **SDSTransaction** and a **type** attribute value of **SDS_TransactionType** (4).

The dynamic attributes for this transaction object type are listed in the following table.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>startTime</td>
<td>TimeStamp</td>
<td>Time at which the transaction started</td>
</tr>
<tr>
<td>status</td>
<td>Integer</td>
<td>Transaction status (see the following table)</td>
</tr>
<tr>
<td>statusText</td>
<td>String</td>
<td>Descriptive text that is displayed for the transaction status</td>
</tr>
<tr>
<td>transactionID</td>
<td>Decimal</td>
<td>Identifier of the transaction</td>
</tr>
<tr>
<td>userName</td>
<td>String</td>
<td>Name of the user</td>
</tr>
</tbody>
</table>

The values of the transaction **status** attribute are represented by one of the **SDSTransactionStates** category global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_TranDeferred</td>
<td>3</td>
</tr>
<tr>
<td>SDS_TranInDoubt</td>
<td>8</td>
</tr>
<tr>
<td>SDS_TranInterrupted</td>
<td>2</td>
</tr>
<tr>
<td>SDS_TranNormal</td>
<td>1</td>
</tr>
</tbody>
</table>
The only value of interest to most users will be SDS_TranInterrupted. When there are active transactions in an interrupted state, read access to persistent objects is not permitted.

**SDS Global Constants**

This section contains the SDS global constants and the categories in which they are defined, as follows.

- JadeDynamicObjectNames Category Global Constants
- JadeDynamicObjectTypes Category Global Constants
- JadeErrorCodesSDS Category Global Constants
- SDSConnectionState
- SDSDatabaseRoles
- SDSEventTypes
- SDSReorgState
- SDSSecondaryState
- SDSStopTrackingCodes
- SDSTakeoverState
- SDSTransactionStates

For details, see the following subsections.

**JadeDynamicObjectNames Category Global Constants**

The global constants listed in the following table define symbolic names for the values of the JadeDynamicObject class name attribute of dynamic objects returned from JadeDatabaseAdmin class query methods.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>String Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_PrimaryName</td>
<td>SDSPrimary</td>
</tr>
<tr>
<td>SDS_SecondaryName</td>
<td>SDSSecondary</td>
</tr>
<tr>
<td>SDS_SecondaryProxyName</td>
<td>SDSSecondaryProxy</td>
</tr>
<tr>
<td>SDS_TransactionName</td>
<td>SDSTransaction</td>
</tr>
</tbody>
</table>
**JadeDynamicObjectTypes Category Global Constants**

The global constants listed in the following table define symbolic names for the values of the `JadeDynamicObject` class `type` attribute of dynamic objects returned from `JadeDatabaseAdmin` class query methods.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_PrimaryType</td>
<td>1</td>
</tr>
<tr>
<td>SDS_SecondaryProxyType</td>
<td>2</td>
</tr>
<tr>
<td>SDS_SecondaryType</td>
<td>3</td>
</tr>
<tr>
<td>SDS_TransactionType</td>
<td>4</td>
</tr>
</tbody>
</table>

**JadeErrorCodesSDS Category Global Constants**

The global constants for the Synchronized Database Service (SDS) error codes that you can use in your own exception handlers, if required, are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>JErr_SdsIllegalOnPrimary</td>
<td>3207</td>
</tr>
<tr>
<td>JErr_SdsIllegalOnSecondary</td>
<td>3206</td>
</tr>
<tr>
<td>JErr_SdsIncompleteJournal</td>
<td>3200</td>
</tr>
<tr>
<td>JErr_SdsInvalidCommand</td>
<td>3205</td>
</tr>
<tr>
<td>JErr_SdsMaxSecondariesExceeded</td>
<td>3210</td>
</tr>
<tr>
<td>JErr_SdsNotInitialized</td>
<td>3201</td>
</tr>
<tr>
<td>JErr_SdsResponseTimeout</td>
<td>3212</td>
</tr>
<tr>
<td>JErr_SdsSecondaryNotAttached</td>
<td>3204</td>
</tr>
<tr>
<td>JErr_SdsSecondaryNotFound</td>
<td>3208</td>
</tr>
<tr>
<td>JErr_SdsTrackerBusy</td>
<td>3211</td>
</tr>
</tbody>
</table>

**SDSConnectionState Category**

The global constants for the Synchronized Database Service (SDS) connection state are listed in the following table. These are used in return values or dynamic object attribute values by the `JadeDatabaseAdmin` class `sdsGetMyServerInfo` method.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_Connected</td>
<td>2</td>
</tr>
<tr>
<td>SDS_Connecting</td>
<td>3</td>
</tr>
<tr>
<td>SDS_ConnectionFailed</td>
<td>4</td>
</tr>
<tr>
<td>SDS_ConnectionStateUndefined</td>
<td>0</td>
</tr>
<tr>
<td>SDS_Disconnected</td>
<td>1</td>
</tr>
</tbody>
</table>
SDSDatabaseRoles Category

The global constants for the Synchronized Database Service (SDS) database roles are listed in the following table. These are used in return values or dynamic object attribute values by the JadeDatabaseAdmin class sdsGetMyServerInfo, sdsGetDatabaseRole, or sdsGetDatabaseSubrole method.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_ROLE_PRIMARY</td>
<td>1</td>
</tr>
<tr>
<td>SDS_ROLE_SECONDARY</td>
<td>2</td>
</tr>
<tr>
<td>SDS_ROLE_UNDEFINED</td>
<td>0</td>
</tr>
<tr>
<td>SDS_SUBROLE_NATIVE</td>
<td>1</td>
</tr>
<tr>
<td>SDS_SUBROLE_RELATIONAL</td>
<td>2</td>
</tr>
</tbody>
</table>

SDSEventTypes Category

The global constants for the Synchronized Database Service (SDS) event types are listed in the following table. These global constants are used in return values or dynamic object attribute values by the JadeDatabaseAdmin class sdsGetMyServerInfo method and in the case of the RPS_SCHEMA_TRANSITION global constant in the SDSEventTypes category, in the userInfo parameter passed to the userNotification callback method that contains one of the RPSTransitionHaltCode category global constants.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPS_SCHEMA_TRANSITION</td>
<td>220</td>
</tr>
<tr>
<td>SDS_CONNECTION_STATE_CHANGE</td>
<td>17386</td>
</tr>
<tr>
<td>SDS_JOURNAL_TRANSFER_STOPPED</td>
<td>17385</td>
</tr>
<tr>
<td>SDS_ROLE_CHANGE_EVENT</td>
<td>22</td>
</tr>
<tr>
<td>SDS_ROLE_CHANGE_PROGRESS</td>
<td>17387</td>
</tr>
<tr>
<td>SDS_TRACKING_HALT</td>
<td>17390</td>
</tr>
<tr>
<td>SDS_TRACKING_STARTED</td>
<td>17389</td>
</tr>
<tr>
<td>SDS_TRACKING_STOPPED</td>
<td>17388</td>
</tr>
</tbody>
</table>

SDSReorgState Category

The global constants for the Synchronized Database Service (SDS) reorganization state are listed in the following table. These are used in return values or dynamic object attribute values by the JadeDatabaseAdmin class sdsGetMyServerInfo method.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_REORG_STATE_NOT_REORGING</td>
<td>1</td>
</tr>
<tr>
<td>SDS_REORG_STATE_OFFLINE_PHASE</td>
<td>5</td>
</tr>
<tr>
<td>SDS_REORG_STATE_REORGING_FILES</td>
<td>4</td>
</tr>
<tr>
<td>SDS_REORG_STATE_RESTARTING</td>
<td>6</td>
</tr>
</tbody>
</table>
Global Constant | Integer Value
---|---
SDS_ReorgStateSeekingApproval | 2
SDS_ReorgStateStarting | 3

**SDSSecondaryState Category**

The global constants for the Synchronized Database Service (SDS) secondary state are listed in the following table. These are used in return values or dynamic object attribute values by the `JadeDatabaseAdmin` class `sdsGetMyServerInfo` method.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_BlockWrite</td>
<td>2</td>
</tr>
<tr>
<td>SDS_JournalSwitch</td>
<td>1</td>
</tr>
<tr>
<td>SDS_StateCatchingUp</td>
<td>1</td>
</tr>
<tr>
<td>SDS_StateDisconnected</td>
<td>0</td>
</tr>
<tr>
<td>SDS_StateReorging</td>
<td>5</td>
</tr>
<tr>
<td>SDS_StateSynchronized</td>
<td>2</td>
</tr>
<tr>
<td>SDS_StateTrackingHalted</td>
<td>4</td>
</tr>
<tr>
<td>SDS_StateTransferHalted</td>
<td>3</td>
</tr>
</tbody>
</table>

**SDSStopTrackingCodes Category**

The global constants for the Synchronized Database Service (SDS) stop tracking are listed in the following table. These are used in return values or dynamic object attribute values by the `JadeDatabaseAdmin` class `getReasonTrackingStoppedString`, `sdsAuditStopTracking`, `sdsGetMyServerInfo`, and `sdsGetSecondaryInfo` methods.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_AuditStopTrackingAll</td>
<td>1</td>
</tr>
<tr>
<td>SDS_AuditStopTrackingNative</td>
<td>2</td>
</tr>
<tr>
<td>SDS_AuditStopTrackingRdb</td>
<td>3</td>
</tr>
<tr>
<td>SDS_ReasonAdminAudited</td>
<td>1</td>
</tr>
<tr>
<td>SDS_ReasonAdminDirect</td>
<td>2</td>
</tr>
<tr>
<td>SDS_ReasonAutoUpgradeMismatch</td>
<td>6</td>
</tr>
<tr>
<td>SDS_ReasonDeltaModeEntered</td>
<td>12</td>
</tr>
<tr>
<td>SDS_ReasonEnablingDbCrypt</td>
<td>13</td>
</tr>
<tr>
<td>SDS_ReasonErrorHalt</td>
<td>8</td>
</tr>
<tr>
<td>SDS_ReasonRestart</td>
<td>10</td>
</tr>
<tr>
<td>SDS_ReasonRpsAdminHalt</td>
<td>4</td>
</tr>
<tr>
<td>SDS_ReasonRpsReorgHalt</td>
<td>9</td>
</tr>
</tbody>
</table>
SDSTakeoverState Category

The global constants for the Synchronized Database Service (SDS) takeover state are listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_ReasonRpsRestart</td>
<td>11</td>
</tr>
<tr>
<td>SDS_ReasonRpsSnapshot</td>
<td>3</td>
</tr>
<tr>
<td>SDS_ReasonTakeover</td>
<td>7</td>
</tr>
<tr>
<td>SDS_ReasonTransition</td>
<td>5</td>
</tr>
</tbody>
</table>

SDSTransactionStates Category

The global constants for the Synchronized Database Service (SDS) transaction states are listed in the following table. These are used in return values or dynamic object attribute values by the JadeDatabaseAdmin class sdsGetMyServerInfo, sdsGetTransactions, or sdsGetTransactionsAt method.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_TranDeferred</td>
<td>3</td>
</tr>
<tr>
<td>SDS_TranInDoubt</td>
<td>8</td>
</tr>
<tr>
<td>SDS_TranInterrupted</td>
<td>2</td>
</tr>
<tr>
<td>SDS_TranNormal</td>
<td>1</td>
</tr>
</tbody>
</table>

These values are the values contained in the userInfo parameter for a role change progress event notification. For details, see "Detecting SDS Role Changes", later in this chapter. See also the sdsInitiateTakeover method of the JadeDatabaseAdmin class.
Developing Applications to Operate in an SDE

This section contains the following topics relating to transaction isolation behavior when developing applications to operate in a Synchronized Database Environment (SDE). For details, see the following subsections.

- Inter-System Event Notifications
  - Persistent Events and Notifications
  - Communicating from a Secondary to a Primary System
- Distinguishing Shadow Transactions from Reader Processes
- Detecting SDS Role Changes
- Journal Transfer Halted Events
- Connection State Change Events
- Tracking Change Events
- Inquiry Access to Secondary Database Servers
- Automatically Starting Server Applications in Secondary Systems

**Note**  JADE licenses are not transferred automatically between databases in an SDE. It is your responsibility to apply new licenses to any existing databases in an SDE. In addition, to ensure proper operation, you must apply the primary license to every secondary.

The secondary starts reorganizing files when it replays an install files audit record, which is audited on the primary as part of the transition (that is, offline) phase of the reorganization and for which the secondary must shut down user applications.

At the onset of a transition, JADE terminates all user applications that are still running, disables signon, and restarts automatically restarted applications at the end of the offline phase of the reorganization.

**Inter-System Event Notifications**

Use the sdeCauseEvent and sdsCauseEvent methods of the Object class for inter-system event notifications. The sdeCauseEvent method combines the actions of the causeEvent and sdsCauseEvent methods, in that subscribers are notified of user events on the local system as well as on SDS secondary or primary systems, where applicable.

In contrast, the causeEvent method would notify subscribers of a user event only on the primary database system and the sdsCauseEvent method only on the secondary database systems.
The two role-dependent usage scenarios for the `sdsCauseEvent` method are as follows.

- From a primary system, to cause persistent events audited by the primary database for replay by secondary database servers.
- From a secondary system, to cause events that are notified to event subscribers on the primary system.

Calling the `sdsCauseEvent` method with the `immediate` parameter set to `true` is applicable only on secondary databases. An exception is raised if you call this method on an SDS primary or on a non-SDS database.

The actions of the `sdsCauseEvent` method are summarized in the following table, which lists the contexts in which the event is caused.

<table>
<thead>
<tr>
<th>Database Role</th>
<th>Transient Target Object</th>
<th>Persistent Target Object, Immediate</th>
<th>Persistent Target Object, Deferred, in Transaction State</th>
<th>Persistent Target Object, Deferred, not in Transaction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Ignored</td>
<td>Exception</td>
<td>Local system</td>
<td>Exception</td>
</tr>
<tr>
<td>Primary</td>
<td>Ignored</td>
<td>Exception</td>
<td>Secondary system</td>
<td>Exception</td>
</tr>
<tr>
<td>Secondary</td>
<td>Ignored</td>
<td>Primary system</td>
<td>Immediately to primary system</td>
<td>Immediately to primary system</td>
</tr>
</tbody>
</table>

The actions of the `sdsCauseEvent` method are summarized in the following table, which lists the contexts in which the event is caused.

<table>
<thead>
<tr>
<th>Database Role</th>
<th>Transient Target Object</th>
<th>Persistent Target Object, Immediate</th>
<th>Persistent Target Object, Deferred, in Transaction State</th>
<th>Persistent Target Object, Deferred, not in Transaction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Process only</td>
<td>Local system</td>
<td>Local system</td>
<td>Local system</td>
</tr>
<tr>
<td>Primary</td>
<td>Process only</td>
<td>Primary system</td>
<td>Primary system and secondary</td>
<td>Primary system only systems</td>
</tr>
<tr>
<td>Secondary</td>
<td>Process only</td>
<td>Secondary system</td>
<td>Secondary system and immediately on the primary system</td>
<td>Secondary system and immediately on the primary system</td>
</tr>
</tbody>
</table>

See also "Causing an Object Event", in Chapter 2.

**Persistent Events and Notifications**

Selected user events, termed *persistent user events*, are audited by the primary database to enable the events to be replayed on the secondary database.

On a primary or non-SDS system, subscribers are notified of non-immediate events at the time the transaction commits (or never, if the transaction aborts). When a persistent user event is replayed on a secondary system, subscribers to the event are notified when the transaction containing the persistent event commits on the secondary system.

Persistent user events can be achieved in two ways, as follows.

- Globally, by setting the `AuditCauseEvents` parameter in the `[SyncDbService]` section of the JADE initialization file to `true`. This parameter is observed only on an SDS primary server and it results in all non-immediate user events caused on persistent objects being converted into persistent user events; that is, they are audited for replay on secondary systems.
- Selectively, by leaving the `AuditCauseEvents` parameter disabled (that is, `false`) and using the `Object` class `sdeCauseEvent` or `sdsCauseEvent` method to explicitly cause persistent user events for secondary replay.

The `userInfo` parameter can be used to associate additional information with the event. This information is passed in the `userInfo` parameter of the `sdeCauseEvent` or `sdsCauseEvent` method to be received by event subscribers in the corresponding `userInfo` parameter of the `userNotification` or `userNotify` callback method.

**Communicating from a Secondary to a Primary System**

Processes executing in an SDS secondary system can communicate with processes executing in the primary, by using SDS event notifications.

When a process executing in a secondary system calls the `Object::sdsCauseEvent` method to cause a user event on a persistent object, the event is notified to subscribers on the primary system (assuming that it is connected) and not to subscribers on the secondary. Conversely, when a process executing in a secondary system calls the `Object::sdeCauseEvent` method to cause a user event on a persistent object, subscribers are notified of user events on the local system as well as on SDS secondary or primary systems, where applicable.

The `sdsCauseEvent` receiver object, also known as the event target, must be a persistent database object that is valid in the primary database at the time the event is caused. All user objects created in the primary database that have not been deleted are candidate event targets.

The `sdeCauseEvent` receiver object, also known as the event target, must be a persistent database object that is valid in the primary or secondary database at the time the event is caused. All user objects created in the primary or secondary database that have not been deleted are candidate event targets. It is also valid to use environmental objects such as `system`, `global`, and `currentSchema` that have the same identity in the primary and secondary database as event targets. On the other hand, it is best to avoid using environmental objects such as `node` and `process`, as these have independent identities and lifetimes between the primary and secondary databases.

To subscribe to events on a primary system caused within a secondary system, use the `beginNotification`, `beginClassNotification`, or `beginClassesNotification` method, identifying the target of the `sdsCauseEvent` and the required event type in the `eventType` parameter.

The `userInfo` parameter can be used to associate additional information with the event. This information is passed in the `userInfo` parameter of the `sdeCauseEvent` or `sdsCauseEvent` method to be received by event subscribers in the corresponding `userInfo` parameter of the `userNotification` or `userNotify` callback method.

**Distinguishing Shadow Transactions from Reader Processes**

As all processes running on the secondary server share the same node object, you can use the `Process` class `type` property to distinguish shadow (replaying) processes on a secondary server from reader processes.

The `type` property value of `5` indicates a shadow process (that is, a process replaying transactions on a secondary server), and can be tested for as follows.

```
if process.type = 5.Charcater then
```

**Detecting SDS Role Changes**

You can call the `System` class `getDatabaseRole` or `getDatabaseSubrole` method to obtain the current database role or subrole for the JADE system in which it is executing or you can register for a system notification that indicates a role change has taken place.
The role change event is caused on the system instance when a database role change has completed. The `eventType` for this event is defined by the `SDS_RoleChangeEvent` global constant in the `SDSEventTypes` category. The `userInfo` parameter contains a value representing the role (that is, the `SDS_RolePrimary`, `SDS_RoleSecondary`, `SDS_SubroleNative`, or `SDS_SubroleRelational` global constant value in the `SDSDatabaseRoles` category).

The role change progress event is caused to notify applications about significant state changes that occur during a takeover operation.

The `eventType` of the role change progress event is defined by the `SDS_RoleChangeProgress` global constant in the `SDSEventTypes` category.

The `userInfo` parameter in a role change progress event contains a value representing the internal state, which can be one of the following `SDSDatabaseRoles` or `SDSTakeoverState` category global constants.

- SDS_RoleUndefined (0)
- SDS_PrimaryRoleActive (1)
- SDS_SecondaryRoleActive (2)
- SDS_TakeoverInitiated (3)
- SDS_HostileTakeoverInitiated (4)
- SDS_RelinquishPrimaryRole (5)
- SDS_RelinquishSecondaryRole (6)
- SDS_TakeoverFailure (7)
- SDS_TakeoverAbandoned (8)
- SDS.WaitForQuietPoint (9)
- SDS.WaitForTakeoverDisposition (10)
- SDS_PrimaryRoleRelinquished (11)
- SDS_SecondaryRoleRelinquished (12)

**Note** When a hostile takeover occurs, the `userInfo` parameter on an SDS secondary contains `SDS_HostileTakeoverInitiated` (4). When a negotiated takeover occurs, the `userInfo` parameter contains `SDS_TakeoverInitiated` (3) for all servers involved in the takeover.

### Journal Transfer Halted Events

The journal transfer halted event is caused on secondary and primary systems when journal transfer has been halted due to some error condition. The `eventType` is defined by the `SDSEventTypes` category `SDS_JournalTransferStopped` global constant.

### Connection State Change Events

The connection state change event is caused on a primary when a secondary connects or disconnects. It is caused on a secondary when the connection to its primary is established or lost. The `eventType` is defined by the `SDSEventTypes` category `SDS_ConnectionStateChange` global constant.

On a primary, the `userInfo` parameter contains the name of the secondary whose connection state changed. The application can then use that name to obtain a secondary proxy and therefore the current connection state.
On a secondary, the `userInfo` parameter contains the value of `connection state` enumerated by the following `SDSConnectionState` category global constants.

- SDS_Disconnected (1)
- SDS_Connected (2)
- SDS_Connecting (3)
- SDS_ConnectionFailed (4)
- SDS_ConnectionStateChangedUndefined (0)

### Tracking Change Events

Tracking change events are caused on a secondary when tracking is enabled or disabled. The `SDSEventTypes` category `SDS_TrackingStopped` or `SDS_TrackingStarted` global constant defines the `eventType` used in the notification.

When the `eventType` is `SDS_TrackingStopped`, the `userInfo` parameter contains zero (0) if you disabled tracking or it contains an error code if an error caused the halting of tracking.

You can call the `JadeDatabaseAdmin` class `sdsAuditStopTracking` method if you want to specify a system event number that is returned in the `userInfo` parameter of your user notification method when an `SDS_TrackingStopped` event occurs because of a programmatic, RPS node, or SDS Administration utility action or tracking is halted because of an error, to notify subscribers that tracking has been disabled.

When the `sdsAuditStopTracking` method is invoked on the primary, it causes a `Stop Tracking` audit record to be written to the current journal. When this record is replayed on an RPS node or on an SDS secondary, tracking halts at that point in the audit trail.

Tracking change events are not notified on the primary. On a primary, the `userInfo` parameter contains the name of the secondary whose transfer has halted. The application can then use that name to obtain a secondary proxy and therefore the error state. On a secondary, the `userInfo` parameter contains an error code that signifies the reason why journal transfer halted.

The `scope` parameter of the `JadeDatabaseAdmin` class `sdsAuditStopTracking` method determines the type of secondary databases that actions the stop tracking command, and is represented by the `SDSSystemStopTrackingCodes` category global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_AuditStopTrackingAll</td>
<td>1</td>
<td>Stops tracking on all JADE and RPS secondary databases</td>
</tr>
<tr>
<td>SDS_AuditStopTrackingNative</td>
<td>2</td>
<td>Stops tracking on JADE native secondary databases</td>
</tr>
<tr>
<td>SDS_AuditStopTrackingRdb</td>
<td>3</td>
<td>Stops tracking on RPS secondary databases</td>
</tr>
</tbody>
</table>

The `reason` parameter of the `JadeDatabaseAdmin` class `getReasonTrackingStoppedString` and `sdsAuditStopTracking` methods determine the reason tracking was disabled, and is represented by the `SDSSystemStopTrackingCodes` category global constants listed in the following table. This value is audited and passed to subscribers to the `SDS_TrackingStopped` system event in the `userInfo` parameter of the associated `userNotify` callback method.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_ReasonAdminAudited</td>
<td>1</td>
</tr>
</tbody>
</table>
## Synchronized Database Service (SDS) Development Considerations

### Global Constants

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDS_ReasonAdminDirect</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonAutoUpgradeMismatch</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonDeltaModeEntered</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonEnablingDbCrypt</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonErrorHalt</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonRestart</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonRpsAdminHalt</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonRpsReorgHalt</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonRpsRestart</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonRpsSnapshot</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonTakeover</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>SDS_ReasonTransition</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

The **SDS_ReasonTakeover** value indicates that tracking stopped during a takeover operation and the **SDS_ReasonRpsReorgHalt** value indicates that tracking stopped at transition. The **SDS_ReasonErrorHalt** value indicates that tracking halted due to an error condition (the error code is saved in the **SDSSecondary** or **SDSSecondaryProxy** dynamic lastErrorCode attribute).

The **SDS_ReasonRestart** value indicates that tracking stopped and then restarted when the connection with the primary was lost. The **SDS_ReasonRpsRestart** value indicates that a condition arose in processing on a primary causing JOM callbackID=2 audit records to be audited for replay purposes, triggering the shut down of applications running from affected schemas when it is replayed on a native or RPS secondary. On an RPS secondary, there is an additional requirement to restart the Datapump application to allow the kernel to close user class agents it may have in use as a result of the replication process. Tracking and the Datapump application are then restarted.

The **journal** and **offset** output parameters of the **sdAuditStopTracking** method contain the journal number and byte offset within the journal of the **Stop Tracking** audit record. These two values together comprise a Log Sequence Number (LSN). When tracking is restarted on the secondary or RPS node, it resumes at the next audit record.

**Note** The main purpose for this in an RPS context is to establish a journal trigger that coincides with a point-in-time on the primary database, to enable establishing a snapshot of the mapped extent in the target database frozen at that time.

The **RPSTransitionHaltCode** category provides global constants that enable you to determine whether the Relational Population Service (RPS) table alter script will be loaded automatically or whether manual action is required from the RDB administrator when a schema instantiation is replayed by an RPS node and the **Datapump** application and database tracking are halted to achieve a schema transition.

When the event **RPS_SchemaTransition** (event type 220) is caused on the system instance, the **userInfo** parameter passed to the **userNotification** callback method contains one of the global constants listed in the following table.

<table>
<thead>
<tr>
<th>Global Constant</th>
<th>Integer Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPS_HaltNoScript</td>
<td>0</td>
<td>Changes do not affect RDB, so no script was generated</td>
</tr>
</tbody>
</table>

---

**Chapter 10  Synchronized Database Service (SDS) Development Considerations**
### Inquiry Access to Secondary Database Servers

This section contains the following inquiry access topics.

- Inquiry Consistency
  - Hiding Effects of Uncommitted Transactions

#### Inquiry Consistency

To provide a consistent database view while the database tracker is redoing updates from the primary system, transactions are isolated.

In particular, as updates are being applied, the effects of updates retain the same transaction-level isolation and atomicity that existed when the transaction was processed on the primary system. This is achieved by hiding effects of uncommitted transactions.

#### Hiding Effects of Uncommitted Transactions

The effects of uncommitted transactions are hidden by maintaining access to the latest committed state of objects. Readers will see only the latest committed state of an object at any time.

To hide the effects of uncommitted transactions, access to uncommitted objects in the three update states is handled as follows.

- **Created object**
  
  A dirty read of a created object causes an exception 4 *(Object not found)* to be raised.

- **Deleted object**
  
  A dirty read of a deleted object returns the object in its undeleted form.

- **Updated object**
  
  A dirty read of an updated object returns the object as it existed prior to the update.

### Automatically Starting Server Applications in Secondary Systems

When a secondary database server is stopped or it crashes when transactions are in progress and it is subsequently restarted, the server remains in a ‘recovery required’ state until all interrupted transactions have been resolved.
Interrupted transactions are resolved when the commit or abort audit records are eventually replayed. While interrupted transactions exist, read-access to the secondary database is not permitted.

In a secondary JADE system, the automatic initiation of server applications specified in the ServerApplication parameter in the [JadeAppServer] section and the ServerApplication parameter in the [JadeServer] section of the JADE initialization file is delayed until read access to the secondary database has been granted. This occurs when:

- The ReadAccessDisabled parameter in the [SyncDbService] section of the JADE initialization file is set to false.
- Journal replay has resolved all interrupted transactions.

**Note** Server applications specified in the ServerApplication parameter in the [JadeAppServer] section and the ServerApplication parameter in the [JadeServer] section of the JADE initialization file that are terminated in a forced takeover operation are not automatically restarted.

See also "Detecting SDS Role Changes", earlier in this chapter.

### Stopping and Starting Server Applications in Secondary Systems

When the instantiation of a versioned schema with or without structural changes is replayed on a secondary database, applications using that schema are stopped and user sign-on is disabled. After the transition phase has completed, server applications specified by the ServerApplication<application-number> parameters in the [JadeAppServer] or [JadeServer] section of the JADE initialization file are started and user-sign-on is enabled.

However, not all changes to classes on the primary system require the schema to be versioned; for example, if a change is made to a class that does not have persistent instances and does not take part in a Relational Population Service (RPS) mapping. This scenario involving a non-versioned schema is considered in further detail in the remainder of this section.

Applications that could potentially use the class must still be stopped and restarted so that they can use the new definition of the class. Applications that could not possibly use the class (for example, an application run from a superschema of the schema in which the class is defined) are not stopped.

When a number of schema loads have changed class definitions but have not required the schema to be versioned, applications are stopped on the secondary system to prevent outdated class definitions being used.

In the absence of a transition phase to identify the end of the sequence of schema loads, that point can be identified and the server applications started and user sign-on enabled in either of the following ways.

- Execute the non-GUI client program on the primary system with the value of the action argument set to auditEnableSecondaryApps. An audit record is written in the journal, which when replayed on the secondary, starts the server applications and enables sign-on. The shortcut to run the non-GUI client program would be as follows:

  ```
  jadclient path=c:\jade\system schema=RootSchema app=JadeReorgApp startAppParameters action=auditEnableSecondaryApps
  ```

  For more information, see "Reorganizing the Database from the Command Line", in Chapter 3 of the JADE Development Environment User's Guide.

- Execute the sdsAuditEnableSecondaryApps method defined in the System class on the primary system. This writes an audit record into the journal. If a reorganization or schema transition is still in progress when the method is executed, a 3418 (Database reorganization in progress) exception is raised.
This chapter covers the following topics.

- Overview
- SOAP-Based Web Services
  - JADE Web Service Provider
  - JADE Web Service Consumer
  - SOAP Versions
    - Defining the Classes, Properties, and Methods for a Web Service
      - Creating a Web Service Class
      - Creating a SOAP Header Class
      - Creating Web Service Methods
    - Using the Web Service Exposure Wizard
      - Adding a Web Service Exposure Definition
      - Displaying a Hierarchical Web Service Exposure Browser
      - Changing a Web Service Exposure Definition
      - Removing a Web Service Exposure Definition
      - Extracting a Web Service Exposure
      - Generating JavaScript for a Web Service Exposure Definition
    - Defining a Web Services Provider Application
      - Generating Web Services Description Language (WSDL)
      - Setting Up Version Control
      - Testing the Web Services Interface
      - Web Service Provider Example
  - Defining a Web Service Consumer Application
    - Using the Web Service Consumer Wizard to Add a Web Service Consumer
    - Displaying a Hierarchical Web Service Consumer Browser
    - Reloading a Web Service Consumer
    - Removing a Web Service Consumer
    - Extracting a Specific Web Service Consumer
    - Generating JavaScript for a Web Service Consumer
- Runtime Web Services Framework
Using Communications Protocols Other than HTTP in your Web Service

Creating a Web Service Gateway

Mapping JADE Primitive Types to XML Schema Types

Web Services Classes

REST-Based Web Services

JadeRestService Class

Creating a REST Service Class

Creating REST Service Methods
  – GET Method Examples
  – POST Method Examples
  – PUT Method Example
  – DELETE Method Example

Defining a REST Services Application
  – Describing Available REST Services

Exceptions in REST Processing

Syntax of a REST Request
  – Parsing JSON Text

Processing a REST Request
  – Deriving the Method Name
  – Passing Primitive Parameters
  – Specifying the Output Format
  – Passing an Object Parameter
  – Passing Additional Information
  – Returning a Value
  – Deleting Transient Objects

Overview

A Web service usually uses HTTP to exchange data. Unlike a Web application, which is typically HTML over HTTP, a Web service is Extensible Markup Language (XML) over HTTP. A client sends a request in XML, and the server responds with an XML response. This XML can be Plain Old XML (POX), which is typically a nonstandard XML that only the client and server can make sense of, or it is standard Simple Object Access Protocol (SOAP).

A Representational State Transfer (REST) API is a service. A REST API differs from SOAP-based Web services in the manner in which it is intended to be used. By using REST, the API tends to be lightweight and embraces HTTP. For example, a REST API leverages HTTP methods to present the actions a user would like to perform and the application entities would become resources these HTTP methods can act on. Although SOAP is not used, messages (requests and responses) are either in XML or JavaScript Object Notation (JSON).
The following sections contain information about the SOAP-based and REST-based Web services functionality that JADE provides.

**SOAP-Based Web Services**

JADE provides a framework that enables you to build distributed JADE applications using the Extensible Markup Language (XML) Web Services architecture.

A Web service is a Uniform Resource Locator (URL) addressable set of functionality that is exposed over a network to serve as a building block for creating distributed applications.

You cannot run Unicode Web service applications.

**Notes** In the first part of this chapter, the term *Web service* specifically refers to an XML-based Web service. JADE supports the Web service provider and Web service consumer. For details, see "JADE Web Service Provider" and "JADE Web Service Consumer", respectively, in the following subsections.

Only the HTTP protocol is directly supported in this release. If you do not want to use the jadehttp communications module provided as part of the Web services framework, you can still invoke the Web service from your own code. (For more details, see "Using Communications Protocols Other than HTTP in your Web Service", later in this chapter.) For example, you could write your own Simple Mail Transfer Protocol (SMTP) handler to receive e-mail and create a SOAP message from this e-mail. You can then create an instance of the Web service class to which the SOAP message can be sent, which will invoke the appropriate method and return a SOAP message as a response. This SOAP message can then be sent back to the user or processed further.

The foundations for Web services are HyperText Transfer Protocol (HTTP), XML, and Simple Object Access Protocol (SOAP, a lightweight HTTP and XML-based protocol that is used for information exchange), which are governed by the World Wide Web Consortium (W3C). As Web services encapsulate the implementation and provide an interface for communicating with the Web service, you can use them as building blocks for applications.

There is no restriction on the granularity of a Web service. It can range from simple components (for example, an order-tracking component published by a shipping company) to large applications (for example, hosted financial applications). You can apply Web services at many different levels of a solution.

Web services can provide convenient access to a static set of information; for example, a Web service can allow a customer to request demographic information for a specified city. Alternatively, you could use Web services to implement highly interactive applications; for example, a travel Web site might make it possible to build an entire vacation itinerary online, by using multiple Web services. A user of that travel application could then use Web services to make hotel and rental car reservations, plan flight itineraries, book flights, and so on.

A Web service can aggregate other Web services to provide a sophisticated set of services; for example, a Web service for a real estate agency can make use of a Web service for a credit verification to facilitate approval of online loan applications. Web services provide:

- **Interoperability**

  As Web services are invoked by using SOAP, which is platform-neutral, you do not have to determine how to build bridges between the Distributed Component Object Model (DCOM), the Common Object Request Broker Architecture (CORBA), and other disparate protocols. Any Web service can interoperate with any other Web service and because Web services communicate by using HTTP and XML, any network node that supports these technologies can both host and access Web services.

- **Multilanguage support**

  As you can write Web services in any language, you do not have to learn new languages or standardize on a single language to create or consume Web services.
Reuse of existing applications

It is easy to expose existing components and libraries as Web services. Many vendors provide tools to make the task of exposing components and libraries even easier. As most companies have a large number of existing components, libraries, and applications, it may be more cost-effective to reuse the functionality in these resources than to reimplement them.

Use of industry-supported standards

The support by all major vendors of technologies that are related to Web services (specifically, HTTP, XML, and SOAP) make it easy for heterogeneous systems to communicate. For example, a component written in JADE and exported as a Web service can easily be used by a Common Gateway Interface (CGI) application that is written in C++ if that application were to make a SOAP request and process the result appropriately.

JADE includes an XML parser, which is based on the Expat XML parser library. The JADE XML parser provides a Document Object Model (DOM) interface and a Simple API for XML (SAX) interface. For details about the XML parser and the RootSchema system classes that support this, see Chapter 12, "Processing XML Documents".

For details about generating a Web service consumer unit test class and stub methods, see "Generating a Web Service Consumer Unit Test Class and Stub Methods", in Chapter 17.

JADE Web Service Provider

One of the important roles in the Web service architecture is that of the Web service provider. The infrastructure that a Web service provider makes available to support and host Web services include the following.

- HTTP protocol handling and authentication services

- A protocol listener, which must be an HTTP listener in JADE

- Because a Web service provider can host multiple Web services, it must also be able to direct the request to an appropriate Web service

- As unknown Web service consumers can access a Web service provider, the Web server must provide basic security services at the protocol level

The JADE Web framework uses Microsoft Internet Information Server (IIS) or Apache as its Web server to provide this functionality. In addition, JADE provides the following.

- A mechanism for classes, properties, and methods to be exposed for a Web service (for details, see "Defining the Classes, Properties, and Methods for a Web Service", in the following section)

- Generation of the Web Services Description Language (WSDL) file from the class, property, and method information (for details, see "Generating Web Services Description Language (WSDL)", later in this chapter)

- A special Web application type (Web Service) and additional application options for using this service (for details, see "Defining a Web Services Provider Application", later in this chapter)

- Application framework for running the service (for details, see "Generating Web Services Description Language (WSDL)", "Setting Up Version Control", and "Testing the Web Services Interface", later in this chapter)

JADE Web Service Consumer

The JADE Web service consumer enables you to access external Web services (including JADE Web services) from within a JADE application. A Web service consumer service cannot be called asynchronously from a JADE application.
Some of the highlights of the JADE Web service consumer feature are as follows.

- Use of a wizard to import Web Services Description Language (WSDL) from WSDL files or a Uniform Resource Identifier (URI)
  
  When a WSDL file is imported into JADE, a subclass of the `JadeWebServiceConsumer` class is created for each service that is defined in the WSDL.

- Setting up Simple Object Access Protocol (SOAP) message requests for the requested Web service method with the appropriate parameters

- Creating the required proxy objects based on the SOAP message response

- SOAP faults are converted to JADE exceptions

- Ability to handle `RPC/Encoded`, `RPC/Literal`, `Document/Literal Bare`, `Document/Literal Wrapped`, and `Document/Encoded` messages

- SOAP header support

- Abstraction of the messaging and transport layers

- Supports HTTP directly, but other protocols can also be used by method reimplementation

For details about defining a Web service consumer application and adding a class to a Web service consumer, see "Defining a Web Service Consumer Application" and "Creating a Web Service Class", respectively, later in this chapter.

**SOAP Versions**

The JADE Web services framework supports SOAP version 1.1 and version 1.2. A JADE Web service consumer creates request messages using the SOAP version that matches the version specified in the imported WSDL.

When you generate the WSDL for a JADE Web service provider you can choose whether the provider expects incoming messages to be formatted according to the SOAP 1.2 standard or the earlier SOAP 1.1 standard. For more details about the specifying the SOAP version, see "Defining a Web Services Provider Application", later in this chapter.

If a Web service provider is running and an incoming message for a SOAP version different from that specified is processed, a Version Mismatch SOAP error is returned and the message is not processed. If the version of the SOAP request message is as expected, the request message is processed and a response is generated using the same SOAP version.

You can import WSDL definitions for both SOAP 1.1 and SOAP 1.2. In that case, a separate `JadeWebServiceConsumer` subclass is created using the service name for each service definition in the WSDL file if the service contains one port definition.

For service definitions containing more than one port definition, a `JadeWebServiceConsumer` subclass is created using the port name for each port definition.

Each port definition is bound to SOAP 1.1 or to SOAP 1.2, so that each consumer processes all messages using this SOAP version, although different `JadeWebServiceConsumer` subclasses can use differing SOAP versions.

When you select a `JadeWebServiceConsumer` subclass in the Class List of the JADE development environment Class Browser, the SOAP level (that is, SOAP 1.1 consumer or SOAP 1.2 consumer) for that class is displayed in the editor pane with the other class details.
Defining the Classes, Properties, and Methods for a Web Service

The JADE Web services framework provides classes that you can subclass for your Web services, and the functionality that enables you to define methods and properties in the your subclasses.

- Creating a Web Service Class
- Creating a SOAP Header Class
- Creating Web Service Methods
  - Specifying Your Web Services Options

For details, see the following subsections.

Creating a Web Service Class

In the Class Browser of your schema, create the ErewhonInvestmentsService class as a subclass of the JadeWebServiceProvider class. (For details, see “Defining Your Own Classes”, in Chapter 3 of the JADE Development Environment User’s Guide.)

To create a Web service provider class

1. Click the tab of the Web Services sheet that is then displayed in the Define Class dialog.

2. On the Web Services sheet, shown in the following image, select the Secure or Not Secure option button in the Secure Service group box to meet your requirements if you do not want your class to use the default security value defined for the application.

3. In this release, only Secure Sockets Layer (SSL) security is provided; that is, the HyperText Transfer Protocol
secure (HTTPS) protocol.

Select the Secure option button if you want to use the HTTPS protocol or the Not Secure option button if you want to use the HyperText Transfer Protocol (HTTP) protocol.

The default value of Default uses the secure setting defined in the application options. (For details, see "Defining a Web Services Provider Application", later in this chapter.)

Note: The TCP protocol for direct Web services communication, which is not secure, works only between JADE systems.

4. If you are setting up SOAP headers, you can specify the direction of SOAP headers in the Soap Header Direction group box. This group box is enabled when you are creating or updating a SOAP header class. For details, see "Creating a SOAP Header Class", in the following section.

By default, the Input option button is selected; that is, the direction is from the Web service consumer to the Web service provider. Select the Output option button if you want the SOAP header direction from the Web service provider to the Web service consumer or select the I-O option button if you want the SOAP header direction from the Web service consumer to the Web service provider and the reverse.

5. If you are adding a subclass to the JadeWebServiceProvider class, check the Must Understand check box if you want the SOAP header to be mandatory for the Web service consumer recipients to process. By default, this check box is unchecked; that is, the SOAP header is optional.

When the SOAP header is mandatory, the consumer of the header entry must obey the semantics conveyed by the fully qualified name of the SOAP header and process correctly to those semantics, or it must fail processing the message. As SOAP headers that must be understood can modify the semantics of their parent or peer headers, the semantics cannot be ignored by those who cannot fully understand them.

6. If you are adding a subclass to the JadeWebServiceProvider class, you can specify the recipient (consumer) of the SOAP header in the Actor text box.

Use this when you do not intend all parts of a SOAP message to be sent to the ultimate destination of the SOAP message but for one or more intermediaries on the message path. Only the consumer of a SOAP header can receive the SOAP header; that is, a consumer of a SOAP header cannot forward the header to the next application in the SOAP message path. The consumer can insert a similar header, which can be provided to another consumer of the SOAP header.

The value of the SOAP header actor is a Uniform Resource Identifier (URI).

The special URI http://schemas.xmlsoap.org/soap/actor/next indicates that the SOAP header is intended for the first SOAP application to process the message. If you do not specify an actor of the SOAP header, the recipient (consumer) of the Web service provider is the ultimate destination of the SOAP message.

7. If you are adding a subclass to the JadeWebServiceConsumer class and you want to change the end-point Uniform Resource Locator (URL) in the WSDL, specify the URL of the consumer in the Service Consumer - Endpoint URL text box.

8. If you do not want to specify any other class options, click the OK button. Alternatively, click the Cancel button to abandon your selections.
The following image shows an example of the Class List of the Class Browser in which the **ErewhonInvestmentsService** class has been created as a subclass of **JadeWebServiceProvider**.

---

**Creating a SOAP Header Class**

SOAP structures a message into two main parts: the headers and the body. SOAP is not only a sender-receiver protocol but it supports the concept of a message passing from a recipient, possibly through one or more intermediaries, and ending up at its destination, more precisely known as the ultimate receiver.

Along the way, intermediaries can perform processing on the message or its side effects. For example, a message can pass through a transaction service (providing a client with guaranteed invocation in the presence of network failures), a security service can sit at an enterprise portal (providing authentication information), and so on.

As intermediaries work primarily on the metadata of the SOAP message, SOAP headers are the ideal place for such data. In addition, they are also a good place to put optional information and a good way to support evolving interfaces. If session handling is enabled in JADE, for example, JADE uses a SOAP header to pass the session id between the service and the client.

For details about specifying the consumer of a SOAP header and that the consumer must understand the semantics of the header, see "Creating a Web Service Class", in the previous section.

To simplify the use of SOAP headers, JADE provides the **JadeWebServiceSoapHeader** class that you can subclass.

**To create a SOAP header class**

1. In the Class Browser of your schema, subclass the **JadeWebServiceSoapHeader** class. (For details, see "Defining Your Own Classes", in Chapter 3 of the **JADE Development Environment User’s Guide**.)

The **Web Services** sheet of the Define Class dialog is then enabled. For details about defining a Web service class, see "Creating a Web Service Class", in the previous section.
The following image shows an example of the Class List of the Class Browser in which the MyHeader class has been created as a subclass of JadeWebServiceSoapHeader, which has a property called count and this is displayed in the Properties List. The value of the count property is included in the SOAP header when the SOAP message is generated.

![Class List](image)

SOAP headers are defined against Web service methods. For details, see the following section.

**Creating Web Service Methods**

You can create Web service JADE methods only for Web service classes.

**Note** Methods declared on the JadeWebService class and its subclasses that are marked as Web service methods cannot have a return type of Any and cannot have parameters of type Any. (See also "webService Option" under "Method Options", in Chapter 1.)

For details about defining a JADE method for a non-Web service class, see "Defining and Compiling JADE Methods and Conditions", in Chapter 4 of the JADE Development Environment User’s Guide.

**To define a Web services JADE method**

1. In the Class List of the Class Browser, click on the subclass of the JadeWebService class or one of its subclasses to which the method is to be added.

2. Select the New JADE Method command from the Methods menu. The Jade Method Definition dialog is then displayed.

3. Uncheck the Web Services check box if you do not want the JADE method to be a Web services method.
When you click the OK button on the Jade Method Definition dialog and the Web Services check box is checked, the signature of your method in the editor pane then includes the webService method option. For example, if you defined a getClients method, the method signature that is created initially is as follows.

```java
getClients(): webService;
```

The following is an example of a getClient method defined in the ErewhonInvestmentsService subclass of the JadeWebServiceProvider class.

```java
getClient(clientName: String): Client webService;
vars
    client: Client;
begin
    client := app.myCompany.allClients[clientName];
    if client = null then
        setError(23, clientName, 'Client does not exist');
    endif;
    return client;
end;
```

You can add or remove the webService method option, which toggles the type of method (that is, to change it from a Web services method to a JADE method, or the reverse).

You can specify any type of parameter (that is, constant, input, io, or output) for a Web services method. For details, see "Parameters", in Chapter 1.

The method does not require a return value because the Web service uses the HTTP protocol (which always requires a response) and a reply is always sent to the Web service consumer. Primitive parameters that are io or output are also sent back in the response as are object parameters that are not constant.

---

**Tip**  
JADE timestamp values use the local time zone. External Web service consumers often expect Coordinated Universal Time (UTC) values and external Web service providers often return UTC values. You may need to convert between UTC and local timestamp values using the localToUTCTime and utcToLocalTime methods of the TimeZone primitive type.

## Specifying Your Web Services Options

When the method selected in the Methods List of the Class Browser is a Web services method, the Web Services Options command in the Methods menu is enabled.

### To specify your Web services options

1. In the Methods List of the Class Browser, select the JADE method of the subclass of the JadeWebService class or one of its subclasses whose Web services options you want to specify.

2. Select the Web Services Options command from the Methods menu. This command is disabled if the selected method is not defined in a subclass of the JadeWebService class or one of its subclasses and the webService method option is not specified for that method.
The Web Services Method Options dialog, shown in the following example, is then displayed.

3. In the list box, select the SOAP header that you want to be part of this Web service message.

   Note  In this release, you cannot set the actor or mustUnderstand attribute of SOAP headers at the method level. The values defined at the SOAP header class level are used.

4. Click the OK button. Alternatively, click the Cancel button to abandon your selection.

Using the Web Service Exposure Wizard

A Web services exposure definition specifies one or more JadeWebServiceProvider subclasses, a collection of JADE classes and properties, and options for the Web service such as the SOAP version and message encoding style.

The exposure is used to generate the WSDL for the consumers who use the Web service.

Note  A Web service exposure definition is extracted or loaded when the schema in which it is defined is extracted or loaded.

To open an Exposure Browser window for a Web services exposure

1. From the Schema Browser, select the Exposures command from the Browse menu.

   An Exposure Browser window is then opened.

2. Click on the tab of the Web Services sheet.

If you have not yet defined a Web service exposure definition, nothing is displayed in this browser. The Exposure Browser provides the Exposure menu, which enables you to maintain your Web service exposure definitions. This menu contains the commands listed in the following table.

<table>
<thead>
<tr>
<th>Command</th>
<th>For details, see…</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Adding a Web Service Exposure Definition</td>
<td>Displays the Web Service Exposure wizard, to enable you to add a new Web Service exposure definition</td>
</tr>
</tbody>
</table>
Adding a Web Service Exposure Definition

Use the Add command from the Exposure menu to invoke the Web Service Exposure wizard, which enables you to define a Web service exposure definition of classes and properties, together with operational specifications about the Web service such as the SOAP version and message encoding style.

Each schema within your JADE database can have a collection of Web service exposure definitions.

The Web Service Exposure wizard is then displayed. You can select one of the following buttons at any time (as appropriate).

- **< Back**, to display the next sheet of the wizard
- **Next >**, to display the next sheet of the wizard
- **Cancel**, to exit from the wizard without saving any changes

For details about the Web Service Exposure definition steps, each contained on a separated sheet of the wizard, see the following subsections.

- **Selecting Classes and Options for Your Web Service Exposure**
- **Selecting Properties for Inclusion in Your Web Service Exposure**

To add a Web service exposure definition

- From the Web service Exposure Browser, select the Add command from the Exposure menu.
The **Classes** sheet of the Web Service Exposure wizard, shown in the following image, is then displayed.

![Web Service Exposure Wizard](image)

**Selecting Classes and Options for Your Web Service Exposure**

The **Classes** sheet of the Web Service Exposure wizard enables you to define the name of the exposure, the **JadeWebServiceProvider** subclasses on which it is based, and some operational specifications about the Web service such as the SOAP version and message encoding style.

**To specify your Web service exposure options**

1. In the **Exposure List Name** text box, specify the name of your new Web service exposure.

2. In the **Select Web Service Classes** list box, select the Web service classes that you want to be part of this application.
Notes You must select one or more Web service classes.

If two JadeWebServiceProvider classes in a hierarchy (for example, SuperProvider and SubProvider) are selected and the resulting WSDL is imported into JADE, the structure is flattened so that every class becomes a subclass of the JadeWebServiceConsumer class.

If two unrelated JadeWebServiceProvider classes are selected (for example, Provider#1 and Provider#2), the Web service methods must have unique names.

Web service methods with the same name that are part of an exposure must have a common superclass containing a method with the same name; otherwise the WSDL generation will fail.

3. Check the Include Session Handling check box if you want to include session handling features. When you select session handling, a SOAP header is automatically generated with a session id for every request. Using session handling gives you access to the currentSession system variable and state maintenance features.

Note Session timeout and minimum response time features work only when session handling is used.

4. Check the Secure Service check box if you want Web services classes to be secure by default.

Note You can override this setting at the Web services class level, if required.

5. Check the HTTP GET Support check box if you want your Web services application to use the HTTP GET protocol. By default, this is unchecked; that is, the HTTP GET protocol support is not enabled.

6. Check the HTTP POST Support check box if you want your Web services application to use the HTTP POST protocol. By default, this is unchecked; that is, the HTTP POST protocol support is not enabled.

7. If you did not check the Bare Format check box in step 10 of this instruction and you want your Web service provider application to use the Remote Procedure Call (RPC) instead of the Document SOAP message style, check the Use RPC check box.

The RPC Format group box is then enabled. If you want to set up a Web service provider exposure with the RPC/Literal format instead of the default RPC/Encoded format, select the Literal option button.

Note In JADE, SOAP messages can be the Document/Literal Bare, Document/Literal Wrapped, Document/Encoded, RPC/Literal, or RPC/Encoded format. By default, JADE uses the Document/Literal or Document/Encoded (if there are cyclic references) message style; that is, the SOAP body simply contains an XML document.

The sender and receiver must agree on the format of the document before the message is sent. As the agreement between the sender and receiver is typically negotiated through literal XML Schema definitions, this combination is referred to as document/literal, which relies on XML Schema to describe exactly what the message looks like.

As RPC is traditionally used in conjunction with the SOAP encoding rules, the combination is referred to as rpcencoded. RPC format messages indicate that the SOAP body contains an XML representation of a method call; for example, the traditional distributed component technologies of DCOM, CORBA, and others.

The RPC style uses the names of the method and its parameters to generate structures that represent the call stack of the method. (For details, see the SOAP 1.2 messaging framework at https://www.w3.org/TR/2007/REC-soap12-part0-20070427). You can then serialize these structures into the SOAP message according to a set of encoding rules.

8. Check the Version Control check box if you want the target namespace to contain the application version.
By default, this is unchecked; that is, no version control check is performed. When version control is enabled, when a request is made the incoming target namespace is compared to the target namespace of the application. If these do not match, a SOAP fault is generated.

9. Check the Use SOAP 1.2 check box if you want your Web services application to use the SOAP 1.2 standard.

   By default, this is unchecked; that is, the SOAP 1.1 standard is used. The SOAP specification defines a standard set of encoding rules for this purpose (see section 5 of the SOAP 1.1 specification) that codify how to map the most common programmatic data structures (for example, structs and arrays) into an XML 1.0 format.

10. If you did not check the Use RPC check box in step 7 of this instruction and you want your Web service provider application to use the Document/Literal Bare format instead of the default Document/Literal Wrapped SOAP message style, check the Bare Format check box.

   **Note** You cannot use the Document/Literal Bare format if you select classes and properties to be exposed that will cause the references to be circular. When defining the classes and properties, you are warned of this, if this is the case.

**Selecting Properties for Inclusion in Your Web Service Exposure**

The Exposure sheet of the Web Service Exposure wizard enables you to select the classes and properties that are included in the exposure.
An example of the Exposure sheet is shown in the following image.

The Web Service Exposure Wizard dialog displays all of the classes and properties that will be exposed to the Web service consumer. By default, all public and read-only properties will be exposed. Use the popup (context) menu for navigation and for property selection.

To change the default setting, select or unselect the properties to meet your requirements. The WSDL generator will generate the schema based on this selection.

When you click the OK button, JADE makes a check for circular references. For example, selecting Company::allClients and Client::myCompany is a circular reference, which results in the display of the warning dialog, which lists the class and property names that may cause circular reference problems.

If you want to stop the current update so that you can modify your selection, click the recommended Cancel button. The detected items are then highlighted in the Web Service Exposure Wizard dialog list box.

**Note** If you click the OK button to proceed with the update, the generated WSDL uses the Document/Encoded message style format rather than Document/Literal Wrapped or Document/Literal Bare format.
Displaying a Hierarchical Web Service Exposure Browser

You can display a hierarchical Web Services Exposure Browser that lists only the classes, properties, constants, and methods in the selected exposure.

To display a hierarchical Web Services Exposure Browser

- From the Web Services sheet of the Exposure Browser, select the Browse command from the Exposure menu.

The hierarchical view of your Web service exposure is then displayed in a browser similar to the Class Browser, as shown in the following image, which is an example of the initial form of the browser.

The initial form of the hierarchy Exposure Browser displays:

- The class list in the upper left pane, containing a hierarchical list of classes in the exposure. Classes shown in:
  - Black are classes not in the exposure but are superclasses of the classes that are in the exposure
  - Green are included in the exposure but they have no property, constant, or method exposed
Blue are those classes in the exposure that have at least one property, constant, or method exposed.

Click on a class, to list the properties, constants, and methods exposed for that class, and to display the usual class information in the editor pane.

The list of properties and constants in the upper middle pane that are exposed for the selected class.

Exposed:
- Properties are displayed using black text
- Constants are displayed using green text, and they are listed after the properties names in the same list

Click on a property or constant to display the usual browser information for the property or constant in the editor pane. For a C# exposure, the external name and type are also displayed.

Right-click on a property to display a popup menu that contains the References, Update References, and Read References commands. Right-click on a constant to display a popup menu that contains the References command.

The methods list in the upper right pane, containing the methods that are exposed for the selected class.

- Click on a method to display the method source in the editor pane. You cannot edit the source in this window.
- Right-click on a method to display a popup menu that contains the References command.

An editor pane, containing a description of the currently selected entity. This text is read-only, and cannot be edited.

**Note** Other than your JADE color preferences, the form does not offer other display options supported by the hierarchy Class Browser or the options that are specified on the Browser sheet of the Preferences dialog.

When you press F4 to search for a class by name, the search results return only the classes in the exposure when this form has focus.

To display a composite view of entities in a versioned schema, select the Show Composite View command in the View menu. For details, see "Toggling the Display of a Composite View of the Hierarchy Exposure Browser".
An example of entities displayed in the composite view of the hierarchy Exposure Browser is shown in the following image.

![Composite View Example](image)

**Toggling the Composite View of the Hierarchy Exposure Browser**

When a schema is versioned, the hierarchical Exposure Browser enables you to toggle the display of a composite view.

- To toggle the display of a composite view in a hierarchical Exposure Browser
  - Select the **Show Composite View** command from the View menu. A check mark is displayed at the left of the menu command when a composite view is selected.

  **Note**  
  This command is enabled only when the schema is versioned.

When the **Show Composite View** command is checked, the current schema is a versioned schema, and the selected schema is the current schema version, the properties, constants, and methods lists display entities that:
Are versioned, including entries for both the current version and the latest version

Have been deleted or removed from the exposure in the latest version

Have been added to the exposure in the latest version

When the Show Composite View command is checked, the current schema is a versioned schema, and the selected schema is the latest schema version, the properties, constants, and methods lists display:

- Entities that are versioned, including entries for both the current version and the latest version

The added and deleted items are not displayed, because it is confusing as to which version of the schema the entries were deleted from or added to.

Note: The composite view uses standard color options that you control using the Preferences dialog.

### Changing a Web Service Exposure Definition

You can modify an existing Web service exposure definition in the current schema.

#### To change a Web service exposure definition

- From the Exposure Browser, select the Change command from the Exposure menu.

The Web Service Exposure wizard is then displayed. For details about the Web Service Exposure wizard steps, see "Adding a Web Service Exposure Definition", earlier in this chapter.

When you are changing an exposure, only classes that have already been selected and classes that are determined as being required (by checking the parameters and return types of the Web service methods) are displayed for selection, so that the list is not cluttered with classes that are not likely to be needed. The Show All check box on the Exposure sheet of the Web Service Exposure wizard displays all classes that were displayed when the service was created.

When you are changing an existing Web service exposure definition, the Exposure List Name text box in the Classes sheet displays the name of your selected Web service exposure definition.

### Removing a Web Service Exposure Definition

From the Exposure Browser, the Remove command from the Exposure menu enables you to remove (delete) the Web service exposure definition that is currently selected.

#### To remove a Web service exposure definition

1. In the Exposure Browser, select the Web service exposure definition that you want to remove.
2. Select the Remove command from the Exposure menu.
3. A message box is then displayed, to enable you to confirm that you want to remove the specified Web service exposure definition.
4. Click the OK button to confirm that the selected Web service exposure definition is to be removed. Alternatively, click the Cancel button to abandon the deletion.

The Exposure Browser is then updated to reflect the removal of the selected Web service exposure definition. (There may be a momentary delay while this updating occurs.)
Extracting a Web Service Exposure

You can extract a Web service exposure as part of the schema in which it is defined, or you can extract only the Web service exposure itself.

To extract a Web service exposure only

1. In the Exposure Browser, select the Web service exposure that you want to extract.
2. Select the Extract command from the Exposure menu. The common Save As dialog is then displayed, to enable you to specify the name and location of your Web service exposure schema. (The common Save As dialog does not enable you to encrypt the saved file.)

The file name defaults to the name of the current Web service exposure, with a .scm suffix. The location defaults to your JADE working directory; for example:

```
s:\jade\test\bin\DemoWebServiceExposure.scm
```

For details about extracting the Web service exposure as part of the schema in which it is defined, see "Extracting Your Schema", in Chapter 10 of the JADE Development Environment User's Guide.

Generating JavaScript for a Web Service Exposure

From the Web Service Consumer Browser, the Generating JavaScript command from the Consumer menu enables you to generate JavaScript for a Web service exposure.

To generate JavaScript for a Web service exposure

1. In the Exposure Browser, select the Web service exposure from which to generate JavaScript.
2. Select the Generate JavaScript command from the Exposure menu. The Web Service JavaScript Generation dialog, shown in the following example, is then displayed.

3. If you want the JavaScript files created in a directory other than the program directory, enter the location in the Output Directory text box. Alternatively, click the Browse button and the common file Open dialog is then displayed to enable you to select the appropriate location.
4. Enter a namespace in the Module Namespace text box. The namespace refers to the root JavaScript variable name that the generated scripts use. For example, if the namespace is Jade, the instruction to call a Web service from JavaScript would be in the following format.

```
Jade.<provider>.<method>({parameters});
```
5. Select a Web service provider application that uses the exposure in the Web service application combo box. This enables the generated JavaScripts to access the correct application. (See also Chapter 4, "Using the Rich Internet Application (RIA) Framework", in the JADE Web Application Guide.)

6. Click the Generate button. Alternatively, click the Cancel button to abandon your selections.

A number of JavaScript (.js) files are created in the specified directory.

Defining a Web Services Provider Application

You can enable your JADE application for Web services when you first define your application or you can access the Define Application dialog at any time to enable Web service access or to maintain your application preferences.

You can override Web application options that you define in the Define Application dialog at the Web services class level, if required.

For details about generating WSDL, see "Generating Web Services Description Language (WSDL)", later in this section.

For more details about the Define Application dialog, see "Defining Applications", in Chapter 3 of the JADE Development Environment User’s Guide. For details about the HTML Documents and JADE Forms sheets of the Define Application dialog, see "Specifying Your HTML Thin Client Options", in Chapter 1 of the JADE Web Application Guide.

To access the Web Options sheet of the Define Application dialog

1. Perform one of the following actions to open an Application Browser window.
   - Click the Browse Applications button from the browse toolbar
   - Select the Applications command from the Browser menu, or press Ctrl+L

2. From the Application menu of the Application Browser, select the Add command to add a new application to your current schema or the Change command to enable Web services access or to maintain the preferences of your current application.

   The Application sheet of the Define Application dialog is then displayed.

3. Select the Web-Enabled or Web-Enabled Non-GUI option in the Application Type combo box, to specify that the application can be accessed from the Web using an HTML thin client.

   The Web Application Type group box and the Web Options sheet are then enabled.

4. In the Web Application Type group box, select the Web Services option button.

5. Click the tab of the Web Options sheet.

   Now that you have enabled your application for Web services, you can specify your access options.
The **Web Options** sheet, shown in the following image, is then displayed.

![Web Options Sheet](image)

6. If you are defining an application for direct Web services, select the **tcp** value in the **Scheme** combo box. As the **tcp** value applies only to direct Web services, retain the default **http** communication protocol value for all other Web service types.

   **Note** The TCP protocol for direct Web services communication works only between JADE systems.

7. In the **Machine Name** text box, specify the machine name or Internet Protocol (IP) address to which the Web service requests should be directed.

   **Note** For direct Web services, the machine name must contain the machine name or the IP address followed by a colon (:) character then by a TCP port number on which this service is offered.

8. In the **Virtual Directory** text box, specify the virtual directory where the **jadehttp** module resides (as defined in IIS or Apache).
For details, see "Configuring JadeHttp for Remote Connections" or "Configuring Apache for Remote Connections", in Chapter 2 of the JADE Installation and Configuration Guide.

9. In the Web Service Exposures Available list box, select the Web service exposure for this application, and then click the right arrow (>) button to move it to the Web Service Exposures to Use list box. The Generate WSDL button is then enabled. Alternatively, double-click on an exposure in one list to move it to the other list.

   **Note** You must select one or more Web service exposures.

   When you resize the Define Application dialog, the Web service exposure list boxes are resized and repositioned to make use of the extra space. For details about defining a Web service exposure, see "Using the Web Service Exposure Wizard", earlier in this chapter.

10. Click the Generate WSDL button. For more details, see the following subsection.

   **Tip** If you want to restrict the number of Web service applications, you can set the LimitPortRange parameter in the [WebOptions] section of the JADE initialization file to true. This prevents your Web service applications from using a port number that exceeds the starting port number plus the number of copies of the application.

   When you set this parameter to true, an application cannot be started when the port limit is exceeded, and a message is output to the jommsg.log file.

### Generating Web Services Description Language (WSDL)

A Web Services Description Language (WSDL) file is generated for each selected Web service exposure from the properties and options defined for that exposure.

The Unicode version of JADE generates UTF8-encoded WSDL files that can be imported by JADE, .NET, and other Web service consumers.

See "Using the Generated WSDL", later in this chapter, for an example of the JADE code defined for the getClient method in the ErewhonInvestmentsService subclass (created in the example under "Creating a Web Service Class", earlier in this chapter).

When you have specified your Web services options and clicked the Generate WSDL button on the Web Services sheet of the Define Application dialog, the common Save As dialog is displayed, to enable you to specify the location and name of the generated file.

The generated WSDL file (of type .wsdl) is based on the following entities.

- The selected Web service classes and its methods
- Classes and selected properties selected for exposure
- Machine name and virtual directory
- Session handling
- Secure service
- Version control
- Selected SOAP headers for the Web service methods
- Use of RPC
- Circular references
Note As changing any of these entities can make a previously generated WSDL incompatible, it is important that you design your Web service carefully and use some form of version control to detect requests from Web service consumers who may be using a WSDL version that is no longer compatible.

Setting Up Version Control

You can set up version control in several ways, including the following.

- Using the JADE version control, which uses the application version and the target namespace to determine version incompatibilities.
- Creating a different Web service class (for example, a subclass) whenever there is a revision, which enables you to run multiple versions simultaneously.
- Creating versions based on method names; for example, if the original method name is `getClient_v1` and the method signature or functionality is changed, the new method name could be `getClient_v2`.

Testing the Web Services Interface

To test your Web services application

1. Set up the application as a Web-enabled application so that the Web Application Monitor window is displayed when you run the application.

   **Note** If your Web-enabled application is not the current active application in the Application Browser, use the Set command from the Application menu to set it to the current active application.

   Alternatively, you can use the Set As Default check box in the Run Application dialog.

2. Ensure that your workstation has the virtual directory specified in step 7 under "Defining a Web Services Provider Application", earlier in this chapter, defined as a virtual directory in Internet Information Services (accessed from the Internet Services Manager command on the Control Panel Administrative Tools submenu). For more details, see "Configuring JadeHttp for Remote Connections" and "Configuring Apache for Remote Connections", in Chapter 2 of the JADE Installation and Configuration Guide.

3. Select the Open Browser command from the Web Application Monitor utility File menu. (For details, see "Monitoring Your Web Sessions", in Chapter 2 of the JADE Web Application Guide.)

Your default browser (for example, Microsoft Internet Explorer) is then started and a browser session connects to the application.
The browser displays a sample home page for your application, as shown in the following example of the Erewhon system.

ErewhonInvestmentsService

Options:
The following operations are supported. To review the Web Services Description File, click here.

- `getAllClients`
- `getClient`
- `getClientNames`
- `updateClient`

In the following images in this subsection, which provide further examples of the Web browser when testing a Web services application, only the form is displayed; that is, the menu, tools, and address bars are not included in the images.

The Web services form is displayed when you click on the `getClient` method. A text box is displayed for each parameter in the method signature. For example, the `getClient` method (described under "Creating Web Service Methods", earlier in this chapter) has one parameter; that is, `clientName`, which has a String value.
Specify a value (for example, **Brian Olsen**) in the **clientName** text box, as shown in the following image, and then click the **Invoke** button.

This action sends the request to JADE as a HTTP POST message and displays the message response, which is a summary of the returned XML document, as shown in the following image.

- **Message Response**:
  The following view provides a summary of the returned XML document. Click [here](#) to view the full XML document.

  - **schema**
On the Web form, click the here hyperlink to display all of the XML document or click the schema hyperlink to display the next level in a new Web browser window. The Web browser form shown in the following image displays the first portion of the full XML document.

**ErewhonWebService**

**Message Response:**

The following view provides a summary of the returned XML document.

Click here to view the full XML document.

- **schema**
  - **getClientResponse**
    - **getClientResult**
      - **address1**: 2834 The Palace
      - **address2**: San Diego
      - **address3**: United States of America
      - **email**: bo@wcl.com
      - **fax**: 642 2930 9393
      - **name**: Brian Olsen
      - **phone**: 1 2 364 6384
      - **website**: www.wcl/olsen.com
      - **allRetailSales**
        - **RetailSale**
          - **mySaleItem**
            - **price**: 6250000.00
          - **timestamp**: 2003-08-11
Not all Web service methods can be tested this way. As methods that require objects as input cannot be tested by using the Web browser interface, write a client application to test this, if required. For example, the following image shows the results of selecting the `updateClient` method (which takes a `Client` object as input) from the Web browser.

You can also display the XML document view (that is, the raw XML from the Web service) when testing your application in the Web browser.
The following image is an example of the first portion of the XML that is displayed when you click the here hyperlink on the Erewhon system sample home page (shown in the first image in this subsection), to indicate that you want to view the Web Services Description Language (WSDL).

```xml
<?xml version="1.0" encoding="utf-8"?>
<schema xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope"/>
<get ClientResponse>
  <get ClientResult>
    <address>2834 The Palace</address>
    <address>San Diego</address>
    <address>United States of America</address>
    <email>bo@wol.com</email>
    <fax>61 2 2930 9393</fax>
    <name>Brian Olsen</name>
    <phone>1 2 3648 8384</phone>
    <webSite>www.wol/olsen.com</webSite>
    <all Retail Sales>
      <my Sale Item>
        <codeNumber>2003</codeNumber>
        <codePrefix>PROP</codePrefix>
        <for Sale Date>1999-12-03</for Sale Date>
        <full Description>Live in complete privacy and luxury miles from civilisation. Land includes helicopter, satellite television and internet connections.</full Description>
      </my Sale Item>
    </all Retail Sales>
  </get ClientResult>
</get ClientResponse>
```

**Notes**  
"Strange" characters that may be displayed when you view the WSDL for your application indicate the base64 equivalent of a binary property photo. In this release, all binary data is converted to base64 format.

Null primitive data values in Web service output are built as `<tag/>` for String primitive types and as `<tag>value</tag>` for all other data types. Null or invalid Date and TimeStamp values are sent as "1900-01-01".

**Web Service Provider Example**

The Web service provider example in this section uses the Erewhon Investments sample system, in which you created the ErewhonInvestmentsService subclass of the JadeWebServiceProvider class. (For details, see "Creating a Web Service Class", earlier in this chapter.)
The following methods are defined in the ErewhonInvestmentsService class.

```java
getAllClients(): CustomerByNameDict;
getClient(clientName: String): Client;
getAllClientNames(): StringArray;
updateClient(client: Client);
```

Based on this class, the methods defined in the class, and the property selections made in the Web Service Exposure Wizard dialog (documented under "Generating Web Services Description Language (WSDL)", earlier in this chapter), JADE generates the following WSDL.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="urn:JadeWebServices/WebShop/22/07/03/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    targetNamespace="urn:JadeWebServices/WebShop/22/07/03/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <xsd:schema elementFormDefault="qualified"
                targetNamespace="urn:JadeWebServices/WebShop/22/07/03/">
      <xsd:element name="getClient">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="clientName" type="xsd:string" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getClientResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="getClientResult" type="tns:Client" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getClientNames">
        <xsd:complexType>
          <xsd:sequence>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getClientNamesResponse">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="getClientNamesResult" type="tns:ArrayOfString" />
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
      <xsd:element name="getClients">
        <xsd:complexType>
          <xsd:sequence>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>
  </types>
</definitions>
```
<xsd:element name="getClientsResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="getClientsResult" type="tns:ArrayOfClient"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="updateClient">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="client" type="tns:Client"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="updateClientResponse">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="updateClientResult" type="tns:Client"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:complexType name="Client">
  <xsd:complexContent mixed="true">
    <xsd:extension base="tns:AddressableEntity">
      <xsd:sequence>
        <xsd:element name="allRetailSales" type="tns:ArrayOfRetailSale"/>
        <xsd:element name="myCompany" type="tns:Company"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ArrayOfRetailSale">
  <xsd:sequence>
    <xsd:element minOccurs="0" maxOccurs="unbounded" name="RetailSale" type="tns:RetailSale"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="RetailSale">
  <xsd:complexContent mixed="true">
    <xsd:extension base="tns:Sale">
      <xsd:sequence>
        <xsd:element name="price" type="xsd:decimal"/>
        <xsd:element name="timeStamp" type="xsd:dateTime"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Sale">
  <xsd:complexContent mixed="true">
    <xsd:extension base="tns:ModelEntity">
      <xsd:sequence>
        <xsd:element name="mySaleItem" type="tns:SaleItem"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
</xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="SaleItem">
   <xsd:complexContent mixed="true">
      <xsd:extension base="tns:ModelEntity">
         <xsd:sequence>
            <xsd:element name="fullDescription" type="xsd:string"/>
            <xsd:element name="photo" type="xsd:base64Binary"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="ModelEntity">
   <xsd:sequence />
</xsd:complexType>
<xsd:complexType name="RetailSaleItem">
   <xsd:complexContent mixed="true">
      <xsd:extension base="tns:SaleItem">
         <xsd:sequence>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="TenderSaleItem">
   <xsd:complexContent mixed="true">
      <xsd:extension base="tns:SaleItem">
         <xsd:sequence>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="Company">
   <xsd:complexContent mixed="true">
      <xsd:extension base="tns:AddressableEntity">
         <xsd:sequence>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="AddressableEntity">
   <xsd:complexContent mixed="true">
      <xsd:extension base="tns:ModelEntity">
         <xsd:sequence>
            <xsd:element name="address1" type="xsd:string"/>
            <xsd:element name="address2" type="xsd:string"/>
            <xsd:element name="address3" type="xsd:string"/>
            <xsd:element name="email" type="xsd:string"/>
            <xsd:element name="fax" type="xsd:string"/>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="phone" type="xsd:string"/>
            <xsd:element name="webSite" type="xsd:string"/>
         </xsd:sequence>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="ArrayOfString">
  <xsd:sequence>
    <xsd:element minOccurs="0" maxOccurs="unbounded" name="String" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ArrayOfClient">
  <xsd:sequence>
    <xsd:element minOccurs="0" maxOccurs="unbounded" name="Client" type="tns:Client"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="JadeSessionHeader" type="tns:JadeSessionHeader"/>

<xsd:complexType name="JadeSessionHeader">
  <xsd:sequence>
    <xsd:element name="sessionId" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>

<message name="getClientSoapIn">
  <part name="inputParameters" element="tns:getClient"/>
</message>

<message name="getClientSoapOut">
  <part name="return" element="tns:getClientResponse"/>
</message>

<message name="getClientJadeSessionHeader">
  <part name="sessionId" element="tns:JadeSessionHeader"/>
</message>

<message name="getClientNamesSoapIn">
  <part name="inputParameters" element="tns:getClientNames"/>
</message>

<message name="getClientNamesSoapOut">
  <part name="return" element="tns:getClientNamesResponse"/>
</message>

<message name="getClientNamesJadeSessionHeader">
  <part name="sessionId" element="tns:JadeSessionHeader"/>
</message>

<message name="getClientsSoapIn">
  <part name="inputParameters" element="tns:getClients"/>
</message>

<message name="getClientsSoapOut">
  <part name="return" element="tns:getClientsResponse"/>
</message>

<message name="getClientsJadeSessionHeader">
  <part name="sessionId" element="tns:JadeSessionHeader"/>
</message>

<message name="updateClientSoapIn">
  <part name="inputParameters" element="tns:updateClient"/>
</message>

<message name="updateClientSoapOut">
  <part name="return" element="tns:updateClientResponse"/>
</message>

<message name="updateClientJadeSessionHeader">
  <part name="sessionId" element="tns:JadeSessionHeader"/>
<portType name="ErewhonInvestmentsServiceSoap">
  <operation name="getClient">
    <input message="tns:getClientSoapIn" /> 
    <output message="tns:getClientSoapOut" />
  </operation>
  <operation name="getClientNames">
    <input message="tns:getClientNamesSoapIn" />
    <output message="tns:getClientNamesSoapOut" />
  </operation>
  <operation name="getClients">
    <input message="tns:getClientsSoapIn" />
    <output message="tns:getClientsSoapOut" />
  </operation>
  <operation name="updateClient">
    <input message="tns:updateClientSoapIn" />
    <output message="tns:updateClientSoapOut" />
  </operation>
</portType>
<binding name="ErewhonInvestmentsServiceSoap" type="tns:ErewhonInvestmentsServiceSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
  <operation name="getClient">
    <soap:operation soapAction="urn:JadeWebServices/WebShop/22/07/03/getClient" style="document"/>
    <input>
      <soap:body use="literal"/>
      <soap:header use="literal" message="tns:getClientJadeSessionHeader" part="sessionId"/>
    </input>
    <output>
      <soap:body use="literal"/>
      <soap:header use="literal" message="tns:getClientJadeSessionHeader" part="sessionId"/>
    </output>
  </operation>
  <operation name="getClientNames">
    <soap:operation soapAction="urn:JadeWebServices/WebShop/22/07/03/getClientNames" style="document"/>
    <input>
      <soap:body use="literal"/>
      <soap:header use="literal" message="tns:getClientNamesJadeSessionHeader" part="sessionId"/>
    </input>
    <output>
      <soap:body use="literal"/>
      <soap:header use="literal" message="tns:getClientNamesJadeSessionHeader" part="sessionId"/>
    </output>
  </operation>
  <operation name="getClients">
    <soap:operation soapAction="urn:JadeWebServices/WebShop/22/07/03/getClients" style="document"/>
    <input>
      <soap:body use="literal"/>
      <soap:header use="literal" message="tns:getClientsJadeSessionHeader" part="sessionId"/>
    </input>
    <output>
      <soap:body use="literal"/>
      <soap:header use="literal" message="tns:getClientsJadeSessionHeader" part="sessionId"/>
    </output>
  </operation>
</binding>
Using the Generated WSDL

This WSDL can then be used by a Web service consumer for generating the proxy classes and invoking the published methods. To create a Web service application, you do not need any knowledge of XML, SOAP, or WSDL. For example, the JADE code for the `getClient` method is as follows.

```java
getClient(clientName: String): Client;
vars
client : Client;
begin
    client := app.myCompany.allClients[clientName];
    if client = null then
        setError(23, clientName, 'Client does not exist');
    endif;
    return client;
end;
```
A sample SOAP message for invoking this method would be as follows.

```xml
<?xml version="1.0" encoding="UTF-8"?>
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:m0="urn:JadeWebServices/WebShop/22/07/03/">
  <SOAP-ENV:Header>
    <JadeSessionHeader>
      <m0:sessionId></m0:sessionId>
    </JadeSessionHeader>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <m:getClient xmlns:m="urn:JadeWebServices/WebShop/22/07/03/">
      <m:clientName>Brian Olsen</m:clientName>
    </m:getClient>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A sample SOAP response for this request would be as follows.

```xml
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
xmlns:xsd=http://www.w3.org/2001/XMLSchema
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <JadeSessionHeader xmlns="urn:JadeWebServices/WebShop/22/07/03/">
      <sessionId>7cbf601166a9240e</sessionId>
    </JadeSessionHeader>
  </soap:Header>
  <soap:Body>
    <getClientResponse xmlns="urn:JadeWebServices/WebShop/22/07/03/">
      <getClientResult>
        <address1>2834 The Palace</address1>
        <address2>San Diego</address2>
        <address3>United States of America</address3>
        <email>bo@wol.com</email>
        <fax>64 2 2930 9393</fax>
        <name>Brian Olsen</name>
        <phone>1 2 3848 8384</phone>
        <website>www.wol/olsen.com</website>
        <allRetailSales>
          <RetailSale>
            <mySaleItem>
              <fullDescription>Live in complete privacy and luxury miles from civilisation. Land includes helipad, satellite television and internet connections.</fullDescription>
              <photo>/9j/4AAQSkZJRgABAgAAZABkAAD//gASQWRvYmUgSW1hZ2VSZWFkef/sABFEdWNreQBAABAAAhAAyAD/7gAQOWRvYmU/AZAAAAB/9sAhAABgYUBgYIBgYIDAgHCaOWCggICQsJCQsOCw0LDhEUhMhJycnJycnJycnAQkICAkKCQsJCQsOCw0LDhEODg4OEMMNQ4Nw/6ruw57t+GlpaWl+CCj8F/2Q==</photo>
              <price>6250000.00</price>
            </mySaleItem>
            <mySaleItem>
              <price>6250000.00</price>
            </mySaleItem>
          </RetailSale>
        </allRetailSales>
      </getClientResult>
    </getClientResponse>
  </soap:Body>
</soap:Envelope>
```
If there is a fault (for example, if there is a version mismatch), the following fault response message is sent.

```xml
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Sender</faultcode>
      <faultstring>Server Error</faultstring>
      <detail>
        <tns:faultDetails xmlns:tns="urn:JadeWebServices/WebShop/22/07/03/">
          <errorCode>11009</errorCode>
          <errorItem>Version Mismatch</errorItem>
          <errorText>
            There is a version mismatch for the requested service -
            Version requested: urn:JadeWebServices/WebShop/22/07/02/
            Current version: urn:JadeWebServices/WebShop/22/07/03/
          </errorText>
        </tns:faultDetails>
      </detail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

### Defining a Web Service Consumer Application

The Web Service Consumer Browser enables you to add and maintain Web service consumers loaded into the current schema.

Each schema in the JADE database can have a collection of Web service consumers, which are imported into JADE as subclasses of the `JadeWebServiceConsumer` class.

When importing a Web service consumer, `StringArray` and `BinaryArray` class entries are prefixed with the name of the consumer. This applies only when importing a consumer; reloading an existing consumer retains existing class names.

You can change the size of `String` and `Binary` array membership of an imported Web service consumer (in the `Length` text box on the `Membership` sheet of the `Define Class` dialog), to specify the maximum expected data length instead of the default value of 128. The specified length is preserved when the consumer is reloaded.
To open a Web Service Consumer Browser

- Select the Web Service Consumer command from the Browse menu in the Schema, Class, or Primitive Types Browser.

A Web Service Consumer Browser window displaying a list of all Web services that have been added to this schema is then opened and the Consumer menu is displayed in the menu bar.

If you have not yet defined a Web service consumer, nothing is displayed in the Web Service Consumer Browser.

The Consumer menu contains the commands listed in the following table.

<table>
<thead>
<tr>
<th>Command</th>
<th>For details, see…</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Using the Wizard to Add a Web Service Consumer</td>
<td>Displays the Web Service Consumer Wizard</td>
</tr>
<tr>
<td>Browse</td>
<td>Displaying a Hierarchical Web Service Consumer Browser</td>
<td>Displays a hierarchical Web Service Consumer Exposure Browser that lists only the classes, properties, constants, and methods in the selected exposure</td>
</tr>
<tr>
<td>Reload</td>
<td>Reloading a Web Service Consumer</td>
<td>Displays the Web Service Consumer Wizard</td>
</tr>
<tr>
<td>Remove</td>
<td>Removing a Web Service Consumer</td>
<td>Following confirmation of the deletion request, deletes the selected Web service consumer</td>
</tr>
<tr>
<td>Extract</td>
<td>Extracting a Specific Web Service Consumer</td>
<td>Displays the Extract dialog</td>
</tr>
<tr>
<td>Generate JavaScript</td>
<td>Generating JavaScript for a Web Service Consumer</td>
<td>Displays the JavaScript Generation dialog</td>
</tr>
</tbody>
</table>

Using the Web Service Consumer Wizard to Add a Web Service Consumer

The Web Service Consumer Wizard enables you to add a Web service consumer by:

- Specifying WSDL File and Consumer Names
- Defining Your Web Service Consumer Mappings
- Using the Imported Web Service Consumer

For details, see the following subsections.

Specifying WSDL File and Consumer Names

The first page of the Web Service Consumer Wizard enables you to select the WSDL file for your Web service consumer and specify the consumer name.

To add a Web service consumer to the current schema

1. Select the Add command from the Consumer menu. The Select WSDL File sheet of the Web Service Consumer Wizard is then displayed.

2. In the WSDL File Name text box, specify the name of the WSDL file on which to base your Web service consumer. The WSDL file must exist.
Alternatively, you can:

- Click the **Browse** button. The common file Open dialog is then displayed, to enable you to select the WSDL source file from the appropriate location.

- Specify a URL (for example, [http://www.webservicex.net/isbn.asmx?WSDL](http://www.webservicex.net/isbn.asmx?WSDL)). JADE then gets the page defined by this URL.

WSDL file details are then displayed in the text box below the **Consumer Name** text box. The **Next** button is enabled when a valid WSDL file is detected.

3. Click the **Validate** button. If the WSDL is valid:

- The caption of the **Validate** button changes to **Next**.

- The name of the WSDL (based on the file name without the directory path and file suffix) is displayed in the **Consumer Name** text box.
The WSDL file details are then displayed in the text box below the **Consumer Name** text box, as shown in the following example.

If the WSDL is invalid (for example, it does not contain methods or the specified location is incorrect), take the appropriate action and then repeat steps 1 through 3.

If you check the **Generate methods for asynchronous calls** check box, methods for consuming the Web service asynchronously are generated in addition to the methods for synchronous execution. For details about running Web services synchronously and asynchronously, see "Using the Imported Web Service Consumer", later in this chapter.

If you check the **Generate new primitive types** check box, the Web service consumer classes and methods generated from the WSDL use the primitive types `Integer64`, `Byte`, and `TimeStampInterval` where appropriate. (These primitive types were not available in earlier implementations of Web services.)

**4.** If you want to change the default consumer name, enter the required name of the consumer in the...
Consumer Name text box.

**Note**: The Consumer Name text box is enabled only after the WSDL is validated, in step 3.

5. Click the **Next>>** button. Alternatively, click the **Cancel** button to abandon your selections.

**Defining Your Web Service Consumer Mappings**

The second page of the Web Service Consumer Wizard enables you to define your mappings for the consumer, as shown in the following image that uses the example Erewhon Investments System provided on the JADE release medium.

The **WSDL Definition** list box at the left of the Define Mappings page displays the WSDL definition. The corresponding **Jade Name Mapping** list box at the right displays the JADE classes, methods, and properties that will be imported into your schema.
JADE names that are displayed in the **Jade Name Mapping** list box in an orange font indicate that the WSDL naming convention is not a valid JADE convention and that has been changed to conform to the JADE naming rules.

You can edit or rename any entity in this list box before you import the Web service consumer class. You can create these classes as subclasses of a class other than **Object** and you can add or remove prefixes of classes, properties, and methods. You can also change entity names individually, by double-clicking on an entry.

Methods are created against the Web service consumer classes that are imported. These are the methods that can be called from your application.

The WSDL file can also contain enumerations or enumeration lists for primitive properties. As JADE does not have the concept of enumeration, a mapping method is created for these properties. These mapping methods check that when the primitive property is set, it conforms to one (or more than one, for lists) of the values specified in the enumeration. If this is not the case, an exception is raised. The **extendedErrorText** property in this exception contains the list of valid values.

You can expand all class nodes to show all of their entities or collapse them to show only the class name itself. An expanded node is displayed with a minus sign (−) and a collapsed node that has features displays a plus sign (+). A node that has no entities has no sign. Alternatively, select the **Collapse All Classes** or **Expand All Classes** command from the popup menu that is displayed when you right-click in the **Jade Name Mapping** list box.

**To define the mappings for your Web service consumer entities**

1. In the **Jade Name Mapping** list box at the right of the dialog:
   a. If you want to create the Web service consumer class as a subclass of a superclass other than **Object**, enter the valid class name in the text box portion of the **Superclass Name** combo box or select the class from the drop-down list. This applies to all classes listed in the **Jade Name Mapping** list box, except for the service classes, which are always a subclass of the **JadeWebServiceConsumer** class.
   b. If you want to apply a prefix to all subclasses, properties, or methods in the imported Web service consumer class, or remove an existing prefix, select the action that you require in the **Modifier** list box, enter the value in the **Value** text box, and then click the **Modify** button. The specified value is then applied to all of the classes, methods, or properties displayed in the **Jade Name Mapping** list box.
   c. To change the name of a specific entity, double-click on the entity that you want to change.
      (Alternatively, select the **Edit Jade Mapping Name** command from the popup menu that is displayed when you right-click on an entity.)

      The entity is then displayed in the editor bar, to enable you to change the name to the required value and then press Enter (if you do not want to change the values of any more entities).

**Tips** By default, the editor bar is not hidden after each update; that is, the editor bar remains visible after updating an entity so that you can perform multiple edit actions by scrolling up or down the list box by using the up (↑) and down (↓) arrow keys. The List Editor message box is then displayed, prompting you to click the **Yes** button if you want to update the selected values or the **No** button if you want to abandon your changes. The editor bar is then positioned on the next entity, and stays visible until you change focus. To hide the display of the editor bar after updating an entity, select the **Hide Editor After Each Update** command from the popup menu that is displayed when you right-click on an entity.

You can change the color of the editor bar by double-clicking the colored portion at the left of the editor bar (which is orange by default) and then selecting the color that you require from the common Color dialog that is displayed. When you click the **OK** button in the Color dialog, the editor bar is then displayed in your selected color.

2. When you have made all required changes, click the **Update** button. Alternatively, click the **Cancel** button to abandon your changes.
The classes, methods, and properties are then created in your schema. Web service consumer classes are always created as subclasses of the JadeWebServiceConsumer class.

**Using the Imported Web Service Consumer**

In the following examples, there are two Web services with the methods for running them synchronously called getClient and getAgent.

When you import the WSDL, you can optionally generate methods to run Web services asynchronously. If you select this option, additional methods are generated with Begin and End appended to the name of the synchronous method; that is, getClientBegin, getClientEnd, getAgentBegin, and getAgentEnd.

The following is an example of the getClient method generated in the ErewhonInvestmentsService class.

```plaintext
getClient(inputParameters:GetClient) : GetClientResponse webService, updating;
/* This code was generated by Jade and cannot be modified
Runtime Version: 99.0.00 */
vars
  any: Any;
begin
  _callAsync := false;
  _methodName := method.name;
  addParameter("inputParameters", inputParameters);
  any := sendRequest("getClient");
  if any = null then
    return null;
  else
    return any.GetClientResponse;
  endif;
end;
```

**Notes** The addHttpHeader method of the JadeWebServiceConsumer class enables you to add, change, or remove HTTP headers. The getHttpHeader method of the JadeWebServiceConsumer class enables you to examine the value of a user-defined HTTP header.

The getHttpHeaderClient method of the JadeWebServiceConsumer class enables you to examine the value of a client HTTP header sent with a Web service request. The getHttpHeaderServer method of the JadeWebServiceConsumer class enables you to examine the value of a server HTTP header sent with a Web service response.

In the following sections, code examples show how to run the Web service methods synchronously and asynchronously.

**Example of Running a Web Service Synchronously**

The following example uses the ErewhonInvestmentsService Web service consumer class to get a Client object by using the synchronous method.

```plaintext
vars
gc   : GetClient;
gcr  : GetClientResponse;
client : Client;
ws    : ErewhonInvestmentsService;
```
begin
  create ws;
  create gc;
  gc.clientName := 'Andrew Fitzpatrick';
  gcr := ws.getClient(gc);
  client := gcr.getClientResult;
  // do some client processing here
epilog
  delete ws;
  delete gc;
end;

When the Web service consumer is deleted, any transient objects created by the Web service are also deleted. If you want to delete the transient objects but retain the Web service, call the `JadeWebServiceConsumer` class `reset` method (for example, `ws.reset`). This call deletes all of the transient objects that were created as a result of the method invocation. For details about adding methods to `JadeWebServiceConsumer` subclasses, see "Creating Web Service Methods", earlier in this chapter.

Example of Running a Web Service Asynchronously

The following example uses the `ErewhonInvestmentsService` Web service consumer class to get a `Client` object and at the same time, get an `Agent` object by using the asynchronous versions of the methods.

vars
gc : GetClient;
gcr : GetClientResponse;
client : Client;
contextc : JadeMethodContext;
ga : GetAgent;
gar : GetAgentResponse;
agent : Agent;
contexta : JadeMethodContext;
ws : ErewhonInvestmentsService;
context : JadeMethodContext;
begin
  // Associate the Web service with an asynchronous worker application
  create ws transient;
  ws.workerApp := "AsyncWorker";
  // Start the first Web service using the appropriate "...Begin" method
  create gc transient;
  gc.clientName := "Andrew Fitzpatrick";
  contextc := ws.getClientBegin(gc);
  // Start the second Web service using the appropriate "...Begin" method
  create ga transient;
  ga.agentName := "Angela Bettersfield";
  contexta := ws.getAgentBegin(ga);
  // Wait for each context to finish before using the '...End' method
  context := process.waitForMethods(contextc, contexta);
  while context <> null do
    if context = contextc then
      gcr := ws.getClientEnd(gc, contextc);
    elseif context = contexta then
      gar := ws.getAgentEnd(ga, contexta);
    endif;
    context := process.waitForMethods(contextc, contexta);
  endwhile;
client := gcr.getClientResult;
agent := gar.getAgentResult;
// Do processing with the client and agent here
epilog
    delete ws;
delete gc;
delete ga;
end;

**Note**  An asynchronous worker application must be running. For details about configuring an application to process asynchronous method calls, see "Asynchronous Worker Applications", in Chapter 16.

When the Web service consumer is deleted, any transient objects created by the Web service are also deleted. If you want to delete the transient objects but retain the Web service, call the `JadeWebServiceConsumer` class `reset` method (for example, `ws.reset`). This call deletes all of the transient objects that were created as a result of the method invocation. For details about adding methods to `JadeWebServiceConsumer` subclasses, see "Creating Web Service Methods", earlier in this chapter.

**Displaying a Hierarchical Web Service Consumer Browser**

The Consumer menu provides the `Browse` command, which enables you to display a hierarchical Web Service Consumer Exposure Browser that lists only the classes, properties, constants, and methods in the selected exposure.

- **To display a hierarchical Exposure Browser**
  1. In the Web Service Consumer Browser, select the consumer whose entities you want to display in a hierarchical browser.
  2. Select the `Browse` command from the Consumer menu.

The Web Service Consumer Exposure Browser for that consumer is then displayed.

For details about browsing a hierarchical view of a Web Service Exposure, see "Displaying a Hierarchical Web Service Exposure Browser", earlier in this chapter.

**Reloading a Web Service Consumer**

The Consumer menu provides the `Reload` command, which enables you to reload a Web service consumer WSDL; for example, when the WSDL definition is changed. For details about renaming entities, see "Defining Your Web Service Consumer Mappings", earlier in this chapter.

- **To reload a Web service consumer WSDL**
  1. In the Web Service Consumer Browser, select the consumer that you want to reload.
  2. Select the `Reload` command from the Consumer menu.
The Warning message box shown in the following image is then displayed.

![Warning message box]

3. To confirm that you want to proceed, click the Yes button. Alternatively, click the No button to abandon the reloading of the Web service consumer WSDL.

The Web Service Consumer Wizard is then displayed. For details about using this wizard, see "Using the Web Service Consumer Wizard to Add a Web Service Consumer", earlier in this chapter.

**Note** When you reload a WSDL, you cannot change the consumer name; that is, the Consumer Name text box on the Select WSDL File sheet is disabled.

### Removing a Web Service Consumer

From the Web Service Consumer Browser, the Remove command in the Consumer menu enables you to remove (delete) the Web service consumer that is currently selected.

**To remove a Web service consumer**

1. In the Web Service Consumer Browser, select the consumer that you want to remove.
2. Select the Remove command from the Consumer menu. Alternatively, press the Delete key.
   
   A message box is then displayed, to enable you to confirm that you want to remove the specified Web service consumer page.
3. Click the OK button in the Confirm Delete message box to confirm that you want to delete the consumer. Alternatively, click the Cancel button to abandon the deletion.

The Web Service Consumer Browser is then updated to reflect the removal of the selected consumer. All classes related to this consumer are also deleted.

Alternatively, you can use the batch JADE Schema Load (jadloadb) executable, the JadeSchemaLoader application in jadclient, jade, or the Application class startApplicationWithParameter method to specify a JADE Command File (JCF) that contains the name of a Web service consumer to be deleted. For details, see "DeleteWebServiceConsumer Command", in the JADE Schema Load User's Guide.

### Extracting a Specific Web Service Consumer

From the Web Service Consumer Browser, the Extract command from the Consumer menu enables you to extract an individual consumer document.

**To extract a Web service consumer**

1. In the Web Service Consumer Browser, select the consumer subclass that you want to extract.
2. Select the Extract command from the Consumer menu. The Web Service Consumer Extract dialog is then
displayed.

The schema file name defaults to the name of the Web service consumer selected in step 1.

3. In the **Schema File Name** text box, change the Web service consumer schema file name prefix, if required.

Alternatively, if you want to extract the Web service consumer to an existing file or you are unsure of the existing file name or location, click the **Browse** button. The common file Open dialog is then displayed, to enable you to select the appropriate file or location, if required.

4. Check the **Encrypt Sources** check box if you want to encrypt the JADE method source code in your extract file.

By default, extracted source code is not encrypted; that is, this check box is unchecked. Source encryption provides security when you release schema extract files, as the source code is not easily visible (for example, when you deploy an application to a third-party). For details, see "Encrypting Schema Source Files", in Chapter 10 of the JADE Development Environment User’s Guide.

**Caution** Ensure that you extract the encrypted schema to a location different from that of your source schema.

If you subsequently load the encrypted schema (for example, for testing purposes), your method source code is lost if you load it into the same database that contains your original source files, as they are not saved during the decryption process.

5. Click the **OK** button. Alternatively, click the **Cancel** button to abandon your selections.

A partial definition schema (**scm**) file is then extracted to the specified directory or to the current directory if you do not specify a directory.

**Generating JavaScript for a Web Service Consumer**

From the Web Service Consumer Browser, the **Generating JavaScript** command from the Consumer menu enables you to generate JavaScript for a Web service consumer.

To generate JavaScript for a Web service consumer

1. In the Web Service Consumer Browser, select the consumer subclass for which you want to generate JavaScript.

2. Select the **Generate JavaScript** command from the Consumer menu.

The Web Service JavaScript Generation dialog, shown in the following example, is then displayed.
3. If you want the JavaScript files created in a directory other than the program directory, enter the location in the **Output Directory** text box. Alternatively, click the **Browse** button and the common file Open dialog is then displayed to enable you to select the appropriate location.

4. Enter a namespace in the **Module Namespace** text box. The namespace refers to the root JavaScript variable name that the generated scripts use. For example, if the namespace is **Jade**, the instruction to call a Web service from JavaScript would be in the following format.

   ```javascript
   Jade.<provider>.<method>(<parameters>);
   ```

5. Click the **Generate** button. Alternatively, click the **Cancel** button to abandon your selections.

A number of JavaScript (.js) files are created in the specified directory.

### Runtime Web Services Framework

The following image shows the Web services framework in a runtime application.

![Web Services Framework Diagram](image)

You can call the **Application::isWebService** method to determine whether a runtime application is a Web services application.

### Using Communications Protocols Other than HTTP in your Web Service

Although the default implementation of the Web services framework uses the HTTP protocol to receive and send messages, JADE provides supporting methods in the **JadeWebServiceProvider** class that enable you to use other communications protocols; for example, SMTP or TCP. These methods and an example of their use are contained in the following subsections.

**Note**  A TCP communication protocol is used to send and receive direct Web services messages.

In addition, the **JadeWebServiceConsumer** class **invoke** method enables you to send the message to your Web service provider using your own communication handlers or dynamically connects to a Web service (that is, without using or importing a WSDL file).
JadeWebServiceConsumer::invoke Method

Call the JadeWebServiceConsumer class invoke method in the following situations.

- If you do not want to use the JADE Web services communications framework, which currently supports only HTTP (for example, if you prefer to use SMTP rather than the HTTP protocol), you can reimplement this method and send the message to your Web service provider using your own communication handlers. The inputMessage parameter holds the SOAP request message. This method returns the SOAP response returned by the Web service provider.

- If you want to dynamically connect to a Web service (that is, without using or importing a WSDL file), reimplement this method. For example, if you have a JadeWebServiceConsumer subclass called DoItMyself, the method in the following example illustrates calling the Amazon Web service dynamically.

```java
vars
    doItMyself : DoItMyself;
    inputMsg  : String;
    outputMsg : String;
begin
    create doItMyself transient;
    doItMyself.setEndpointURL('http://soap.amazon.com/onca/soap2');
    inputMsg := ; //soap request here
    outputMsg := doItMyself.invoke(inputMsg);
    // outputMsg will now contain the response from the Web service
    // provider or a SOAP fault raised by the JADE Web services
    // framework (for example, if the connection failed)
    epilog
        delete doItMyself;
end;  

Note In this case, you will be dealing directly with SOAP messages.
```

JadeWebServiceProvider::initialize Method

When the default HTTP implementation is not used, no Web service application is run. However, you must still define a Web service application and set up the appropriate options. In your application code, create an instance of your JadeWebServiceProvider subclass and then set up that instance with your options by calling the JadeWebServiceProvider class initialize method, specifying the name of your Web-enabled application in the appName parameter.

The Web service options are then set up. If the application specified in the appName parameter does not exist or it is not a Web service application, this method returns false.

JadeWebServiceProvider::processMessage Method

The JadeWebServiceProvider class processMessage method takes the value of the message parameter as input (which is assumed to be a SOAP message), calls the relevant Web service method, passing it the necessary parameters, and returns the result of the processing as a SOAP message. If the incoming message is not a SOAP message, an exception is raised. Similarly, if the method name or the parameters are not valid, an exception is raised. It is your responsibility to trap this exception and take whatever action is necessary.

Use the Exception class createSOAPMessage method to transform this error into a SOAP message by returning a string representing a SOAP fault message. The receiver is the exception that was raised.
Example of the Use of the Additional JadeWebServiceProvider Class Methods

The following example uses the Erewhon Investments sample system to demonstrate the use of the JadeWebServiceProvider class initialize and processMessage methods.

```plaintext
vars
  soapMsg : String;
  ws      : ErewhonInvestmentsService;
  out     : String;
begin
  // incoming message is hard coded here, but it could be sourced
  // from anywhere
  soapMsg := '<?xml version="1.0" encoding="utf-8"?>
    <soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
    <soap:Body>
      <getClient xmlns="urn:JadeWebServices/WebShop/"
                 <clientName>Andrew Fitzpatrick</clientName>
    </soap:Body>
  </soap:Envelope>';
create ws;
ws.initialize('WebShop');
out := ws.processMessage(soapMsg);
// return the result to the caller
epilog
  delete ws;
end;
```

The `out` variable in this example will contain a SOAP message that has been UTF8 encoded. The user application can process this message further or return it to the caller as it is.

Creating a Web Service Gateway

You can provide a gateway Web service; that is, develop a Web service that is a contact point for other Web services.

All Web service requests are sent to the gateway Web service application. In the most-general case, the Web service requests are handled by separate JADE systems on separate computers.

The gateway Web service application provides no services of its own, and it simply redirects requests provided by the CustomerWebService service to WebApp1 running on host1 and requests provided by the ProductWebService service to WebApp2 running on host2.

Setting up a Gateway Application

To set up a gateway Web service application

1. In your gateway schema, create a Web service provider class that will function as the gateway Web service.
2. Do not define any Web service methods for this class.
3. Define a Gateway Web service application in the usual way. (For details, see "Defining Applications", in Chapter 3 of the JADE Development Environment User's Guide.)
4. Extract the WSDL for the Web service and reload it into the same schema. (You will need to change the name of the Web service consumer class.)

**Setting up a Web Service Application for CustomerWebService requests**

To set up the WebApp1 application to respond to CustomerWebService requests

1. In your customer Web service schema, create a Web service provider class called **CustomerWebService**.
2. Define Web service methods required for **CustomerWebService**.
3. Define a **WebApp1** Web service application in the usual way. (For details, see "Defining Applications", in Chapter 3 of the JADE Development Environment User’s Guide.)
4. Extract the WSDL for the **CustomerWebService** Web service.

**Editing the CustomerWebService WSDL**

The WSDL must be edited so that the initial request is sent to the **Gateway** application on host0 rather than to the **WebApp1** application on host1.

The original WSDL fragment in the following example has the values to be modified displayed in bold.

```xml
<service name="CustomerWebService">
  <port name="CustomerWebServiceSoap" binding="tns:CustomerWebServiceSoap">
    <soap:address location="http://host1/jade/jadehttp.dll?WebApp1&serviceName=CustomerWebService"/>
  </port>
</service>
```

The modified WSDL fragment in the following example has the modified values displayed in bold.

```xml
<service name="CustomerWebService">
  <port name="CustomerWebServiceSoap" binding="tns:CustomerWebServiceSoap">
    <soap:address location="http://host0/jade/jadehttp.dll?Gateway&serviceName=CustomerWebService"/>
  </port>
</service>
```

**Reimplementing Application::setEndpointForWebService on the Gateway**

All requests go through the gateway application, which then redirects requests to the systems that provide the appropriate Web services.

The following example shows a reimplementation of the **setEndpointForWebService** method on the **Application** class with logic to call the appropriate Web service application.

```java
vars
  // GatewayConsumer is a JadeWebServiceConsumer class in current schema
ws: GatewayConsumer;
begin
  create ws;
  if className = "CustomerService" then
    // CustomerWebService is a JadeWebServiceProvider class in the
```
// schema to which the request will be redirected
ws.setEndpointURL("http://host1/jade/jadehttp.dll?WebApp1&"
   & "serviceName=CustomerWebService");
elseif className = "ProductService" then
   // ProductService is a JadeWebServiceProvider class in the
   // schema to which the request will be redirected
      & "serviceName=ProductWebService");
endif;
// The JadeWebServiceConsumer::invoke method redirects the request
return ws.invoke(message);
epilog
   delete ws;
end;

Web Service Gateway Performance Considerations

There are performance considerations when using a Web service gateway, in particular:

- As calls are synchronous, your application design needs to ensure that the gateway provider does not become a bottleneck for the requests. (You could have several gateway providers, to reduce this.)

- The SOAP message passing between the gateway provider and the Web service consumer incurs the overhead of a string copy.

- For better performance, consider using direct TCP/IP Web services between the gateway consumer and the actual Web service provider applications.
Mapping JADE Primitive Types to XML Schema Types

The following diagram represents the XML built-in data type hierarchy.

```
Built-in Datatype Hierarchy

  anyType
  ------------------
  all complex types
  |                   |
  | anySimpleType     |
  -------------------
  duration dateTime time date qYearMonth qYear qMonthDay qDay qMonth
  |
  boolean base64Binary hexBinary float double anyURI QName NOTATION

  string
  |
  normalizeString

  token
  |
  nonPositiveInteger long nonNegativeInteger

  language Name NMTOKEN negativeInteger int unsignedLong positiveInteger

  NCName NMTOKENS
  |
  ID IDREF ENTITY

  IDREFS ENTITIES
```

The mapping of JADE primitive types to XML types is listed in the following table.

<table>
<thead>
<tr>
<th>JADE Attribute</th>
<th>XML Simple Type</th>
<th>Examples (delimited by commas)</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>string</td>
<td>Confirm this is electric</td>
</tr>
<tr>
<td>Character</td>
<td>byte</td>
<td>-1, 126</td>
</tr>
<tr>
<td>JADE Attribute</td>
<td>XML Simple Type</td>
<td>Examples (delimited by commas)</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Binary</td>
<td>base64Binary</td>
<td>GpM7</td>
</tr>
<tr>
<td>Integer</td>
<td>long</td>
<td>-1, 12678967543233</td>
</tr>
<tr>
<td>Decimal</td>
<td>decimal</td>
<td>-1.23, 0, 123.4, 1000.00</td>
</tr>
<tr>
<td>Real</td>
<td>double</td>
<td>-INF, -1E4, -0, 0, 12.78E-2, 12, INF, NaN</td>
</tr>
<tr>
<td>Boolean</td>
<td>boolean</td>
<td>true, false, 1, 0</td>
</tr>
<tr>
<td>Time</td>
<td>time</td>
<td>13:20:00.000, 13:20:00.000-05:00</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>dateTime</td>
<td>1999-05-31T13:20:00.000-05:00</td>
</tr>
<tr>
<td>Date</td>
<td>date</td>
<td>1999-05-31</td>
</tr>
</tbody>
</table>

**Web Services Classes**

This section summarizes the following JADE Web service classes and the properties and methods defined in those classes, if applicable.

- **JadeWebService Class**
  - **JadeWebServiceConsumer Class**
  - **JadeWebServiceProvider Class**
  - **JadeWebServiceSoapHeader Class**

- **JadeSOAPException Class**

For details about these classes and the `Application::isWebService` method, see Volume 1 and Volume 2 of the *JADE Encyclopaedia of Classes*.

**JadeWebService Class**

The **JadeWebService** class is the transient class that defines behavior of JADE Web services. The methods defined in the **JadeWebService** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>isNilItem</td>
<td>Returns true if the item was specified as having a nil state in the XML message being processed</td>
</tr>
<tr>
<td>setError</td>
<td>Specifies the Web service provider error code, item, and text of the SOAP error</td>
</tr>
</tbody>
</table>

For details, see "**JadeWebService Class**", in Chapter 1 of the *JADE Encyclopaedia of Classes*.

**JadeWebServiceConsumer Class**

A subclass of the **JadeWebServiceConsumer** class is created for each Web service consumer loaded into JADE. The properties defined in the **JadeWebServiceConsumer** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains the…</th>
</tr>
</thead>
<tbody>
<tr>
<td>characterConversionException</td>
<td>Whether a Web service response on an ANSI system contains non-ANSI characters</td>
</tr>
<tr>
<td>Property</td>
<td>Contains the...</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>handleCharConversionException</td>
<td>Whether an exception is raised on an ANSI system if a Web service response contains non-ANSI characters</td>
</tr>
<tr>
<td>logStatistics</td>
<td>Value that specifies whether Web service consumer message statistics are logged</td>
</tr>
<tr>
<td>password</td>
<td>Web service consumer user authentication password, if required</td>
</tr>
<tr>
<td>proxyHostName</td>
<td>Host name of the proxy server for the Web service consumer, if required</td>
</tr>
<tr>
<td>proxyPassword</td>
<td>Web service consumer user authentication password for proxy servers, if required</td>
</tr>
<tr>
<td>proxyUsername</td>
<td>Web service consumer user authentication identifier for proxy servers, if required</td>
</tr>
<tr>
<td>soapHeaders</td>
<td>List of SOAP headers sent by the Web service consumer</td>
</tr>
<tr>
<td>soapRequest</td>
<td>Outgoing SOAP message sent to the Web service provider</td>
</tr>
<tr>
<td>soapResponse</td>
<td>SOAP message response received from the Web service provider</td>
</tr>
<tr>
<td>timeout</td>
<td>Number of milliseconds after which the Web service times out if a response has not been received</td>
</tr>
<tr>
<td>unknownHeaders</td>
<td>Contains an array of any SOAP headers that were part of the response but could not be processed</td>
</tr>
<tr>
<td>userName</td>
<td>Web service consumer user authentication identifier, if required</td>
</tr>
<tr>
<td>workerApp</td>
<td>Name of a worker application configured to process a Web service request asynchronously</td>
</tr>
</tbody>
</table>

The methods defined in the **JadeWebServiceConsumer** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addHttpHeader</td>
<td>Adds, changes, or removes HTTP headers from a Web service consumer request</td>
</tr>
<tr>
<td>getEndpointURL</td>
<td>Returns the name of the end-point URL to which the Web service consumer request is sent</td>
</tr>
<tr>
<td>getHttpHeader</td>
<td>Returns the value of a specified user-defined HTTP header</td>
</tr>
<tr>
<td>getHttpHeaderClient</td>
<td>Returns the value of a client HTTP header specified by the value of the key parameter</td>
</tr>
<tr>
<td>getHttpHeaderServer</td>
<td>Returns the value of a server HTTP header specified by the value of the key parameter</td>
</tr>
<tr>
<td>getLastStatistics</td>
<td>Returns statistics relating to the last Web service consumer SOAP message</td>
</tr>
<tr>
<td>getTimeouts</td>
<td>Returns the timeout values in milliseconds for connect, send, and receive messages, respectively</td>
</tr>
<tr>
<td>invoke</td>
<td>Sends the message to your Web service provider using your own communication handlers or it dynamically connects to a Web service (that is, without using or importing a WSDL file)</td>
</tr>
<tr>
<td>invokeAsync</td>
<td>Sends the message asynchronously to your Web service provider using your own communication handlers or it dynamically connects to a Web service (that is, without using or importing a WSDL file)</td>
</tr>
</tbody>
</table>
**Method** | **Description**
--- | ---
invokeAsyncWithVerb | Sends the message and the specified verb asynchronously to your Web service provider using your own communication handlers or it dynamically connects to a Web service (that is, without using or importing a WSDL file)
invokeWithVerb | Sends the message and the specified verb to your Web service provider using your own communication handlers or it dynamically connects to a Web service (that is, without using or importing a WSDL file)
processReply | Processes the result of a Web service request (that is, a SOAP message) and sets up transient objects for further processing by your application
reset | Deletes all transient objects created by the Web service consumer when making a Web service request
sendRequest | Sets up the SOAP message for a Web service request and sends the message to the Web service provider
setEndpointURL | Dynamically changes the URL to which the Web service request is sent
setTimeouts | Sets the timeout values for connect, send, and receive messages, respectively

For details, see "JadeWebServiceConsumer Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

**JadeWebServiceProvider Class**

The **JadeWebServiceProvider** class maintains all Internet service provider information.

The properties defined in the **JadeWebServiceProvider** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleteTransientReturnType</td>
<td>Specifies whether the transient object return type from a Web service method is deleted when processing is complete</td>
</tr>
<tr>
<td>incomingMessage</td>
<td>Contains the incoming SOAP message string from the Web service consumer</td>
</tr>
<tr>
<td>rawXML</td>
<td>Specifies whether the Web services framework does any further XML processing of the data that is returned from a Web service method</td>
</tr>
</tbody>
</table>

The methods defined in the **JadeWebServiceProvider** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getServerVariable</td>
<td>Returns HyperText Transfer Protocol (HTTP) header information for your Web service request</td>
</tr>
<tr>
<td>initialize</td>
<td>Sets up the appropriate options for the specified Web service application when the default HTTP implementation is not used</td>
</tr>
<tr>
<td>processMessage</td>
<td>Calls the relevant Web service method and returns the result of the processing as a SOAP message</td>
</tr>
<tr>
<td>processRequest reply</td>
<td>Processes Web service requests received from a Web service consumer</td>
</tr>
<tr>
<td>reply</td>
<td>Executed when a request is received from a Web service consumer</td>
</tr>
</tbody>
</table>

For details, see "JadeWebServiceProvider Class", in Chapter 1 of the JADE Encyclopaedia of Classes.
JadeWebServiceSoapHeader Class

The **JadeWebServiceSoapHeader** class defines the behavior of SOAP headers in Web service provider applications.

The properties defined in the **JadeWebServiceSoapHeader** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>actor</td>
<td>Contains the URL of the SOAP header recipient</td>
</tr>
<tr>
<td>mustUnderstand</td>
<td>Specifies whether it is mandatory for the recipient of the SOAP header to process the header</td>
</tr>
</tbody>
</table>

For details, see "JadeWebServiceSoapHeader Class", in Chapter 1 of the **JADE Encyclopaedia of Classes**.

JadeSOAPException Class

The **JadeSOAPException** class defines the behavior of SOAP exceptions that are raised in Web service provider applications. For details, see "JadeSOAPException Class", in Chapter 1 of the **JADE Encyclopaedia of Classes**.

See also:

- **Exception** class **createSOAPMessage** method, in Chapter 1 of the **JADE Encyclopaedia of Classes**
- "JadeErrorCodesWebService Category", in Appendix A of the **JADE Encyclopaedia of Primitive Types**
- Web services error messages in the range 11000 through 11052 in "Error Messages and System Messages", in the **JADEMsgs.pdf** file

REST-Based Web Services

REST-based Web services are implemented using HTTP. They offer a light-weight alternative to the original SOAP and WSDL-based Web services.

REST works with resources that are identified with a Uniform Resource Identifier (URI). A resource represents a static or dynamically-generated Web page. REST resources are named with nouns as part of the URI rather than verbs; for example, `/customers` rather than `/getCustomers`.

To use REST services, a client sends an HTTP request using the GET, POST, PUT, or DELETE verb.

The traditional HTTP error messages (for example, **200 - OK** and **404 - Not found**) can be used to indicate whether a request is successful. If a request is successful, information can be returned in Extensible Markup Language (XML) or JavaScript Object Notation (JSON) format. REST services exception handling identifies which part of the service is responsible for an unsuccessful request and returns to the client an HTTP error **400** (bad request from the user) or **500** (server error). A **400** error is returned, for example, if the:

- URL is empty
- JSON or XML syntax is invalid
- Called method is not found
- Called method is protected or it is a type method
- Data for a method parameter is invalid for its type
Data does not match the method signature

- Type of the object passed does not match the method parameter object type

A server error is returned for other failures such as a logic exception.

Session handling is not performed, so there is no timeout of connections. Additionally, information is not retained between requests from a client. If that is required, it must be provided by the application developer.

For details about the JadeJson class, which is a transient-only Object subclass that provides standalone JSON functionality that is independent of the Representational State Transfer (REST) Application Programming Interface (API), see Volume 1 of the JADE Encyclopaedia of Classes. The JadeJson class enables you to create, load, unload, and parse JSON in the same way you can with XML.

JadeRestService Class

A transient instance of a subclass of a JadeRestService class is created by each REST services application and used by each REST services message that is received. The processRequest method is called on this object, passing the message details in the URL. The method decodes the URL and any objects passed in XML or JSON format, and calls the required method on the same JadeRestService object. The result returned by the method is encoded into XML or JSON, as requested. The reply method is then called, passing the string to be returned to the client.

The constants defined in the JadeRestService class are summarized in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Integer Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputFormat_Json</td>
<td>0</td>
</tr>
<tr>
<td>OutputFormat_Json_NewtonSoft</td>
<td>2</td>
</tr>
<tr>
<td>OutputFormat_Xml</td>
<td>1</td>
</tr>
</tbody>
</table>

The properties defined in the JadeRestService class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains the …</th>
</tr>
</thead>
<tbody>
<tr>
<td>httpStatusCode</td>
<td>HTTP status code error returned to the client</td>
</tr>
<tr>
<td>objectsToBeDeleted</td>
<td>Array for non-shared transient objects to be deleted when the REST method has completed</td>
</tr>
</tbody>
</table>

The methods defined in the JadeRestService class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createVirtualDirectoryFile</td>
<td>Passes files created by a JADE application to the jadehttp library</td>
</tr>
<tr>
<td>deleteVirtualDirectoryFile</td>
<td>Deletes specified files from the virtual directory used by the jadehttp library</td>
</tr>
<tr>
<td>getOutputFormat</td>
<td>Returns an Integer value that represents the output format</td>
</tr>
<tr>
<td>getServerVariable</td>
<td>Returns the specified HTTP header information for your REST service request from the Internet Information Server (IIS)</td>
</tr>
<tr>
<td>isVDFilePresent</td>
<td>Returns true if the specified file is present in the virtual directory used by the jadehttp library</td>
</tr>
</tbody>
</table>
Creating a REST Service Class

The JADE REST framework provides the JadeRestService class that you can subclass for your REST services. (For details, see “Defining Your Own Classes”, in Chapter 3 of the JADE Development Environment User’s Guide.)

Creating REST Service Methods

You can create REST service methods only for a JadeRestService subclass. For details about defining a JADE method, see "Defining and Compiling JADE Methods and Conditions", in Chapter 4 of the JADE Development Environment User’s Guide.

The following subsections contain examples of REST service methods to handle GET, POST, PUT, and DELETE requests, which could be defined in a JadeRestService subclass in your schema.

The output parameter type is not appropriate for a REST service method and should not be specified. A method with this parameter type will cause exception 11103 (A parameter of the called method is usage output) at run time.

A parameter with the Any type is initialized to the string value from the URL.

A method return type of Any is encoded according to the type of data returned. If the returned value is an unassigned Any, exception 11111 (Unrecognised response from Rest service exception) is raised because the XML or JSON type is unknown.

When manually testing JADE REST service methods, the REST service is initialized if it has not been already, except if the application is already attached to a JADE Web Service Manager, in which case exception 11126 (A Rest Service method was called but the service was never initialised) is raised. In addition, if the JadeRestService class reply method is called and its processing is not associated with a received web message, exception 11127 (JadeRestServices.reply was called but there is no web message to reply to) is raised. To avoid exception 11127 when manually testing REST service processing, you should re-implement the reply method on the JadeRestService subclass that is being used, and ensure that the reply method does not call inheritMethod.

Notes A method called as the end point of a REST service request cannot be a type method (that is, it must be an instance method).

If access is protected or read-only of a property of an input object to a REST service call, REST does not set the property.

GET Method Examples

The following method returns a Customer object in XML or JSON in response to a GET request in which the customer identifier is supplied.

```java
getCustomer(pId: Integer): Customer updating;
vars
customer: Customer;
```
begin // allCustomers is keyed on the customer id
customer := app.myRoot.allCustomers.getAtKey(pId);
if customer = null then
    // Setting HTTP status optional - you could simply return a 'null' customer
    self.httpStatusCode := 404;
    return null;
else
    // Make an object to return and avoid returning references
    return customer.cloneSelf(true);
endif;
end;

**Note** If you return an object that has a reference to the another object, the related object is also returned as part of the XML or JSON response. This can result in an excessively-large response.

The following method returns a customer’s name in response to a GET request in which the customer identifier is supplied.

getCustomerName(pId: Integer): String;
vars
customer: Customer;
begin // allCustomers is keyed on the customer id
customer := app.myRoot.allCustomers.getAtKey(pId);
if not customer = null then
    return customer.name;
else
    return "";
endif;
end;

**POST Method Examples**

The following method creates a customer in response to a POST request in which the data for the customer is provided as primitive type parameters.

postCustomer(pName: String; pAddress: String);
vars
customer: Customer;
begin
    beginTransaction;
    create customer;
    // Properties are set from the primitive parameters
    customer.name := pName;
    customer.address := pAddress;
    customer.myRoot := app.myRoot;
    commitTransaction;
end;

The following method creates a customer in response to a POST request in which the data for the customer is provided in the request body as an object in XML or JSON format.

postCustomer(pCust: Customer);
// pCust is a transient object created from
// XML or JSON before the method is invoked
The following method creates a customer in response to a POST request in which the data for the customer is provided as parameters that are aggregated into a ParamListType entity.

```haskell
postCustomer(params: ParamListType);
vars
customer: Customer;
begin
beginTransaction;
create customer;
// Properties are set from the parameters in the list
customer.name := app.getParamListTypeEntry(1, params).String;
customer.address := app.getParamListTypeEntry(2, params).String;
customer.myRoot := app.myRoot;
commitTransaction;
end;
```

PUT Method Example

The following method updates an existing customer in response to a PUT request.

**Note** One or more parameters are used to identify the Customer object to be updated. The remaining parameters are used to update the object.

```haskell
putCustomer(pId: Integer; pName: String; pAddress: String);
vars
customer: Customer;
begin
// Identify customer to be updated using pId
customer := app.myRoot.allCustomers.getAtKey(pId);
if not customer = null then
// Update customer using pName and pAddress
beginTransaction;
customer.name := pName;
customer.address := pAddress;
commitTransaction;
endif;
end;
```

DELETE Method Example

The following method deletes a specified customer in response to a DELETE request.

```haskell
deleteCustomer(pId: Integer);
begin
// Delete customer with specified id
```
beginTransaction;
delete app.myRoot.allCustomers.getAtKey(pId);
commitTransaction;
end;

**Defining a REST Service Application**

The REST services application is defined from the Define Application dialog in the standard way. For more details, see "Defining Applications", in Chapter 3 of the *JADE Development Environment User’s Guide*.

On the Application sheet, select **Rest Services** or **Rest Services, Non-Gui** as the Application Type.
On the **Web Options** sheet, select your **JadeRestService** subclass and specify the required IIS connection information.

The **Exclude Protected Properties** check box enables you to specify whether the application definition includes or excludes protected properties from serialization and object construction. When this check box is:

- Unchecked (the default), protected properties are included in any serialization of the object and will be included when constructing the object from the received XML or JSON structure.

- Checked, protected properties are not included in any serialization of the object and are ignored when constructing the object from the received XML or JSON data.

The setting of this check box is loaded when the REST service is initiated, and any change to the check box setting does not take effect until the REST service application is restarted.

**Note**  
Read-only properties are always excluded from object construction from the received XML or JSON data.
The generated description includes only the classes and the superclasses referenced by the specified parameter values and return types of the communications methods of the JadeRestService subclass selected for the application. Note, however, that no class description is included if those methods return only Any or a primitive type.

In the jadehttp.ini file, add an [application-name] section to enable clients to connect to the JADE REST services application. Set the parameter values to match the configuration information you specified on the Define Application dialog.

```
[RestApp]
ApplicationType=RestServices
TcpConnection=localhost
TcpPort=45000
```

For more details about configuring jadehttp for remote connections, see "[application-name] Section", in Chapter 2 of the JADE Installation and Configuration Guide.

**Describing Available REST Services**

You can generate a description of the REST services that are offered by your REST services application, by clicking the Generate Description button on the Web Options sheet of the Define Application dialog.

The description includes only the classes and the superclasses referenced by the specified parameter values and return types of the communications methods of the JadeRestService subclass selected for the application. Note, however, that no class description is included if those methods return only Any or a primitive type.

The XML file contains the following information.

- Application name and required URL.
- Each available resource.
  - HTTP verb; for example, GET, PUT, POST, or DELETE.
  - Resource identifier; for example, Customer.
  - Required parameter names and types.
  - Type of object to be supplied in JSON or XML.
- Each class referred to by the signatures of the resource methods.
  - Name of the class.
  - Name and C# type of each property.

The following table maps JADE property types to C# types.

<table>
<thead>
<tr>
<th>JADE Type</th>
<th>C# Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary</td>
<td>Byte []</td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>Byte</td>
<td>Byte</td>
</tr>
<tr>
<td>Character</td>
<td>Char</td>
</tr>
<tr>
<td>Date</td>
<td>DateTime</td>
</tr>
<tr>
<td>JADE Type</td>
<td>C# Type</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal</td>
</tr>
<tr>
<td>Integer</td>
<td>Int32</td>
</tr>
<tr>
<td>Integer64</td>
<td>Int64</td>
</tr>
<tr>
<td>JadeBytes</td>
<td>Byte[]</td>
</tr>
<tr>
<td>HugeStringArray</td>
<td>String[]</td>
</tr>
<tr>
<td>Point</td>
<td>String format &lt;integer&gt;, &lt;integer&gt;</td>
</tr>
<tr>
<td>Real</td>
<td>Double</td>
</tr>
<tr>
<td>String</td>
<td>String</td>
</tr>
<tr>
<td>StringUtf8</td>
<td>String</td>
</tr>
<tr>
<td>Time</td>
<td>DateTime</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>DateTime</td>
</tr>
<tr>
<td>TimeStampInterval</td>
<td>TimeSpan</td>
</tr>
<tr>
<td>TimeStampOffset</td>
<td>DateTime (with UTC offset set)</td>
</tr>
</tbody>
</table>

The following description file is not an exposure, but it would be of value to a REST client developer building requests and handling responses.

```xml
<Application name="RestApp"
    uri="http://localhost/jade/jadehttp.dll/"
    example="http://localhost/jade/jadehttp.dll/Person.json?RestApp">

<resources>
  <method name="DELETE" id="Customer">
    <request>
      <param name="pId" type="int" />
    </request>
    <response type="empty" />
  </method>

  <method name="GET" id="Customer">
    <request>
      <param name="pId" type="int" />
    </request>
    <response type="Customer" />
  </method>

  <method name="GET" id="CustomerName">
    <request>
      <param name="pId" type="int" />
    </request>
    <response type="String" />
  </method>

  <method name="POST" id="Customer">
    <request>
      <param name="pName" type="String" />
      <param name="pAddress" type="String" />
    </request>

```

DevRef - 2018.0.01
Exceptions in REST Processing

An exception that occurs during the execution of a JadeRestService method results in a Fault object being returned formatted in the style specified in the URL. When the output format is:

- **JSON** (Microsoft JSON), the response is formatted as follows.

  ```json
  {
    "_type":"Fault",
    "errorCode": "<error number>",
    "errorItem": "<Exception type>",
    "errorText": "<exception description>"
  }
  ```

- **JSONN** (NewtonSoft JSON), the response is formatted as follows.

  ```json
  {
    "$type":"Fault",
    "errorCode": "<error number>",
    "errorItem": "<Exception type>",
    "errorText": "<exception description>"
  }
  ```

The following example displays the JSON XML that is returned if exception 1035 (*String too long*) is raised.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Fault>
  <errorCode>1035</errorCode>
  <errorItem>NormalException</errorItem>
</Fault>
```
The XML style is required because JSON does not identify the name of the entity being returned.

There are a number of exceptions that are specific to the processing of a REST request. The following example displays the XML that is returned if the JSON request contains the wrong number of parameters; in this example, two primitive parameters were supplied instead of one.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Fault>
  <errorCode>11106</errorCode>
  <errorItem>The signature of Method 'RestService::getClient' does not match the url data provided: (2) primitive parameters.</errorItem>
  <errorText>The signature of the called Rest Services method does not match the supplied url entities</errorText>
</Fault>
```

For more details about the exceptions that are specific to JADE REST requests, see "11100 Through 11199 - JADE Rest Service Errors", in the JADE Error Messages and System Messages document.

## Syntax of a REST Request

A REST request is sent from a client as an HTTP verb (GET, POST, PUT, or DELETE), followed by the URL of the resource. The syntax is similar to that of other types of JADE Web-enabled applications.

```
Verb IIS server URL/jadehttp.dll/path[.xml|json|jsonn]?app_name[&extra_info]

  <------------------------->  <------------------------->  <------------------------>
  first part                second part                third part
```

The following JADE REST service request retrieves information in JSON format for a customer with an identifier of 123.

```
GET http://localhost/jade/jadehttp.dll/customer/123.json

  <------------------------->  <-------------->
  first part                second part
```

The first part of the URL is the path to the jadehttp.dll file.

```
http://localhost/jade/jadehttp.dll
```

In this example, the IIS host is the local machine and jade is an alias defined in IIS for the physical directory that contains the jadehttp.dll file.

The second part of the URL contains the following.

- Identifier of the resource, which in this example is `customer`.

  The `JadeRestService` method that is invoked for a GET request on the resource `/customer` is obtained by converting the HTTP verb to lowercase (get) and appending the name with the first letter capitalized (Customer), resulting in the method name `getCustomer`.

- Each additional URL path level is a parameter passed to the called method. Each string value is converted to the required method parameter type. An exception is raised if the data is invalid or there is a mismatch in the number of parameters.

A GET request for `/customer/123` would result in a `getCustomer(123)` method call; that is, the `getCustomer` method would require the first parameter to be of the `Integer` type.
A GET request for /customer/Clark Kent would result in a getCustomer("Clark Kent") method call; that is, the getCustomer method would require the first parameter to be of the String type.

Note: REST requests must be URL-encoded before the request is sent, so that /customer/Clark Kent would become /customer/Clark%20Kent.

A GET request for /customer/Clark Kent/Smallville would result in a getCustomer("Clark Kent", "Smallville") method call; that is, the getCustomer method would require the type of the first and second parameters to be String.

URL path levels separated by the slash character (/) are used to pass primitive parameters. An object parameter is passed as XML or JSON as the body of the data received. You can pass one object parameter only in a REST service request.

A ParamListType parameter can be used in the method signature to receive multiple path parameters from the URL but it must be the last parameter of the JADE method. All parameters passed for a ParamListType parameter are assumed to be strings.

You can include the output format of the data at the end of the path information.

- /customer/123.xml returns customer information in Microsoft XML format
- /customer/123.json returns customer information in Microsoft JSON format
- /customer/123.json returns customer information in Newtonsoft JSON format

If the output format is not specified (/customer/123), data is returned in Microsoft JSON format.

The third part of the URL is the query string. It contains the name of the JADE REST services application. In the following example, the JADE REST services application is called RestApp.

GET http://localhost/jade/jadehttp.dll/customer/123.json?RestApp

For a GET request, you can append an ampersand character (&) followed by the XML or JSON for an object, if the REST services method has an object parameter in the signature.

For a POST, PUT, or DELETE request, you can supply the XML or JSON for an object in the request body, if the REST services method has an object parameter in the signature.

The type of the object is specified by the type of the parameter in the signature of the JadeRestService method. The information for that object is passed in the query string after the ampersand character (&) in XML or JSON format.

See also "Parsing JSON Text".

Parsing JSON Text

Standard JSON syntax can include the type of the objects to create; for example:

```json
/> "__type":"Customer", .in Microsoft JSON format
/> "$type":"Customer", .in Newtonsoft (JSONN) format
```

This "__type" special type tag must appear as the first entry following the { or [ symbol that begins the object contents description in Microsoft format. In Newtonsoft, the "$type" tag can appear after the object reference tag; for example:

```
"id":"12", "$type":"Customer",
```
In Newtonsoft, the type can also include a namespace such as "MyNameSpace.Customer", which JADE will ignore.

If the JSON does not include the type tag, the type of the object is assumed to be the implied type of the expected object, for example, the type of the method parameter to be populated or the type of the property reference.

If the JSON includes the type tag, the type must be the implied type or a subclass of the implied type. If it is not, an exception is generated.

When creating JSON from an object using the JadeJson class, the type tag is inserted only when the class is not specified; that is, these tags are output only when the object is a subclass of the specified type. For example, if a reference property is defined to be a Customer and the reference is to a subclass SpecialCustomer, the tag is inserted. Similarly, if a collection is encoded to JSON and a member of the collection is a subclass of the membership type, the tag is inserted.

**Processing a REST Request**

When a message arrives, a transient instance of the JadeRestService subclass is created, if it does not already exist. The processRequest method is then called on this instance. If you reimplement this method, you should always call inheritMethod to complete the processing.

**Note** To avoid exception 11127 (JadeRestServices.reply was called but there is no web message to reply to) when manually testing REST service processing, re-implement the reply method on the JadeRestService subclass that is being used, and ensure that the reply method does not call inheritMethod. For more details about manually testing REST service processing, see "Creating REST Service Methods", earlier in this chapter.

When the processRequest method is called, the path part of the URL (for example, /customer/123) is parsed to construct the method name and the primitive type parameters. If the method signature includes an object parameter, the information for the object must be provided as an XML or JSON script at the end of the query string.

The steps in the processing are described in more details in the following sections.

**Deriving the Method Name**

The first level of the path (customer) is modified by converting the first character to uppercase (Customer). The HTTP verb for the request GET, PUT, POST, or DELETE is converted to lowercase and prepended to Customer, resulting in the name of the method to be called, which is getCustomer, putCustomer, postCustomer, or deleteCustomer.

An exception is raised if the method does not exist on the JadeRestService subclass.

**Passing Primitive Parameters**

Subsequent levels of the path are converted into parameters that are validated, converted to the correct type, and then passed to the method.

Primitive parameters for the called method must be in the URL path in the order that they appear in the method signature. For example, if the signature of the method was getCustomer(pld: Integer; pTime: Time), a GET request for the resource /customer/123/12:45 would result in a call to getCustomer(123, "12:45".Time).

JADE accepts dates in both of the following formats.

<table>
<thead>
<tr>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>/Date(value{+time-zone-offset})</td>
<td>/Date(148824000000+1200)</td>
</tr>
<tr>
<td>yyyy-mm-dd</td>
<td>2017-03-25</td>
</tr>
</tbody>
</table>
A `ParamListType` parameter can be used in the method signature to receive multiple path parameters from the URL, but it must be the last parameter of the JADE method. All parameters passed for a `ParamListType` parameter are assumed to be strings.

### Specifying the Output Format

You can specify the format of the data returned by the REST service at the end of the path information, as shown in the following examples.

- `/customer/123.xml` returns customer information in Microsoft XML format.
  
  The XML format is as expected by the Microsoft `DataContractSerializer` class for an `Object` or a `TimeStampInterval` primitive return value. While other primitives return values are formatted for the Microsoft `XmlSerializer`, `XmlSerializer` does not support the `TimeSpan` type. The output includes identifiers for each object included and references to already included objects, which enables the returned data to contain circular references and multiple references to the same object.
  
  The XML returned for an object lists the properties in alphabetical order within each class of the hierarchy, working from the highest superclass down through each subclass.

- `/customer/123.json` returns customer information in Microsoft JSON format.
  
  The JSON format is as expected by the Microsoft `DataContractJsonSerializer` class. This format type does not support circular references or multiple references to the same object in the returned data. An exception is raised if a circular reference or multiple references to the same object are detected. This format is the default if no format was specified.

- `/customer/123.jsonn` returns customer information in Newtonsoft JSON (JSONN) format.
  
  The JSONN format is as expected by the Newtonsoft JSON class software. It differs from Microsoft in the structure, tags, and the format of some primitive types. The output includes identifiers for each object and references to already included objects, which enables the returned data to contain circular references and multiple references to the same object.

If the output format is not specified (for example, `/customer/123`), data is returned in Microsoft JSON format.

### Passing an Object Parameter

Only one object parameter can be defined for a REST services method, and it can appear in any position in the signature except after a `ParamListType` parameter. If the method signature includes an object parameter, the information for the object must be provided in XML or JSON format. The method can contain references to other child objects that were also passed in the XML or JSON.

In the following `POST` request, an object is passed in XML format.

```
POST http://localhost/jade/jadehttp.dll/customer?RestApp
```

### XML-Formatted Object

An XML-formatted object is identified by its header.

```
<?xml version="1.0" encoding="UTF-8"?>
```

The XML is parsed to create a transient instance of the specified type, which is passed to the REST service method. When the method has completed execution, the transient object and any child objects that were created are deleted.
Classes or properties referenced in the XML that do not exist in the schema are ignored. For a property that does exist, an exception is raised if the passed value in the XML is of the wrong type.

The following XML creates a Customer object.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Customer>
    <name>Clark Kent</name>
    <address>Smallville</address>
</Customer>
```

The XML can indicate that the passed object is null. If the base object is not null, the object must be of the type required by the parameter in the JadeRestService subclass. If it is not the required type, an exception is raised.

If an XML header is not found, the text is searched for the first opening brace character { for a non-Collection parameter type, or for the first opening brace { character or bracket [ character for a Collection type. If such a character is found, it is assumed that the object is in JSON.

**JSON-Formatted Object**

In the following POST request, an object is passed in JSON.

```
POST http://localhost/jade/jadehttp.dll/customer?RestApp
```

The following JSON creates a Customer object.

```
{"name":"Clark Kent","address":"Smallville"}
```

The JSON is parsed and a transient instance of the parameter type is created and populated. As the class name is not specified in JSON, if you pass the wrong object, the entire content is ignored unless both object types have identical property names.

Classes or properties referenced in the JSON that do not exist in the schema are ignored. For a property that does exist, an exception is raised if the passed value in the JSON is of the wrong type.

When the method has completed execution, the transient object is deleted.

**Null Object**

You can pass a null object to a REST request by adding the string null to the request body.

**Passing Additional Information**

After the REST services application name, you can add an ampersand character (&) followed by additional information that could be used in the processing of the REST service request, as follows.

```
POST http://localhost/jade/jadehttp.dll/customer?RestApp&additional_information
```

To process the additional information, you must reimplement the processRequest method in your JadeRestService subclass; otherwise the additional information is ignored.

**Returning HTTP Header Information**

Call the getServerVariable method of the JadeRestService class if you want to return the specified HTTP header information for your REST service request from the Internet Information Server (IIS). This method must be called during the processing of a REST service message; for example, from a re-implementation of the JadeRestService class processRequest method. Calling the method when a message is not being processed results in null always being returned.
The method in the following example returns the IP address of the REST service as determined by IIS.

```javascript
processRequest(httpIn: String; queryStr: String; pathIn: String; methodType: String) updating;
var
  str : String;
begin
  str := self.getServerVariable("ALL_HTTP");
  inheritMethod(httpIn, queryStr, pathIn, methodType);
end;
```

As the `var` parameter is IIS-dependent, it is therefore subject to change. Refer to the `ServerVariables` function in your Internet Information Services (IIS) documentation for details.

Common server environment variables, documented in the IIS documentation under the `ServerVariables` function, include those listed in the following table.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Returns…</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP_ACCEPT_LANGUAGE</td>
<td>A string describing the language to use for displaying content</td>
</tr>
<tr>
<td>HTTP_USER_AGENT</td>
<td>A string describing the browser that sent the request</td>
</tr>
<tr>
<td>HTTPS</td>
<td>ON if the request came in through a secure channel (SSL) or it returns OFF if the request is for a non-secure channel</td>
</tr>
<tr>
<td>REMOTE_ADDR</td>
<td>IP address of the remote host making the request</td>
</tr>
<tr>
<td>SERVER_NAME</td>
<td>Host name, DNS alias, or IP address of the server as it would appear in self-referencing URLs</td>
</tr>
<tr>
<td>SERVER_PORT</td>
<td>Port number to which the request was sent</td>
</tr>
<tr>
<td>URL</td>
<td>Base portion of the URL</td>
</tr>
</tbody>
</table>

The method must be called on the same node as the application. If you call the method from a server method and the application is not running on the server, a 31039 error (Connection invalid invocation) occurs when trying to access the TCP/IP connection, and error 1242 (A method executing in another node was aborted) is reported to the REST service.

### Returning a Value

The `reply` method in the `JadeRestService` class is called and is passed the string constructed from the value returned by the REST services method, if the method returns a value. The string is encoded according to the format requested by the client and sent to the client.

If you reimplement this method, you should always call `inheritMethod` to complete the processing.

If the method returns an object:

- The entire object tree referenced is encoded into XML or JSON, as required.

  **Note** If you return an object that has a reference to another object, the related object is also returned as part of the XML or JSON response. This can result in an excessively large response.

- All property values are returned, including nulls.

If the method returns a string that is already encoded in XML format (that is, starting with `<xml>`), the XML string is sent as it is, without further encoding.
Deleting Transient Objects

The `processRequest` method deletes the following non-shared transient objects.

- Any non-shared transient objects created from the passed XML or JSON.
- The returned value if it is a non-shared transient object.
- Any objects added to the `objectsToBeDeleted` collection by user logic. Note that these objects must be non-shared transient objects.

A returned shared transient object is not deleted on completion of the REST Services processing. It would be unsafe to do so, because JADE cannot be certain of whether that is the intention and JADE would have to go into transaction state to do so.

**Note** Anything added to the `JadeRestService` class `objectsToBeDeleted` collection is expected be to non-shared transients, and any other object lifetime will cause the logic to fail because the logic is not in transient state.
Chapter 12  Processing XML Documents

This chapter covers the following topics.

- **Overview**
- **Jade XML Tree Model**
  - XML Tree Architecture
  - Using the XML Tree Model
  - Parsing an XML Tree Document
    - Processing Extra Whitespace
    - Creating XML Tree Documents
    - Retrieving Information from XML Tree Documents
    - Traversing XML Documents
    - Modifying XML Tree Documents
    - Writing XML Tree Documents
    - Handling XML Tree Exceptions
- **JADE XML Parser Model**
  - JADE XML Parser Model Architecture
  - Using the XML Parser Model
- **JADE XML Classes**

**Overview**

JADE XML allows JADE systems to parse, create, navigate, manipulate, and write XML documents. JADE provides the following alternative API models for the processing of XML documents.

- **JADE XML Tree**, which models an XML document as a tree of JADE objects.

  JADE XML Tree is a read-write API that can parse existing XML documents and create new ones. Each XML document is represented as a *JadeXMLDocument* object. Documents are searched, queried, and updated by invoking methods on this *JadeXMLDocument* object and the objects that it contains. (For more details, see "Jade XML Tree Model", later in this chapter.)

**Tip**  Although JADE XML Tree model is very convenient when you require random access to widely separated parts of the original document, it is resource-intensive compared with the JADE XML Parser model.

- **JADE XML Parser**, which models the parser itself rather than the document.

  JADE XML Parser is an event-driven read-only API that reads an XML document serially and feeds the content to the application by a callback interface. The parser is represented by a *JadeXMLParser* object.
The application creates a subclass of JadeXMLParser and implements the JadeXMLParser callback methods on the subclass. (For more details, see “JADE XML Parser Model”, later in this chapter.)

**Tip** Although the JADE Parser model is fast and memory-efficient because it does not have to store the entire document in objects, JADE XML Parser applications can be harder to design and code because you normally need to develop your own data structures to hold the content from the document.

Both JADE XML Tree and JADE XML Parser models use the same underlying parsing engine, which checks the document to ensure that it is well-formed but it does not validate the document against a schema or document type definition.

### XML Syntax

XML (the Extensible Markup Language) is a text-based markup language for documents containing structured information and is used in a variety of applications, particularly for data interchange on the Internet.

A document uses tags enclosed in angle brackets (&lt; &gt;) to identify data. A well-formed document must be syntactically correct; for example, each opening tag must have a corresponding closing tag and tags must nest properly.

A valid document implies that the document conforms to a specified grammar; for example, an XML Schema or Document Type Definition (DTD) can specify the kinds of tags and their ordering. This is optional.

### Jade XML Tree Model

XML has a simple markup syntax that allows you to store information as a tree of tagged nodes.

Consider the XML document `library1.xml`, shown in the following example, that describes the contents of a library.

```xml
<?xml version="1.0"?>
<!--An example XML document-->
<library>
  <book isbn="0-246-13655-3">
    <title>Mystery</title>
    <author>Peter Straub</author>
  </book>
  <book isbn="1-876590-17-3">
    <title>False Memory</title>
    <author>Dean Koontz</author>
  </book>
</library>
```

In this example:

- The first line of the file is the XML declaration with version information.
- The second line defines a comment node.
- The next line contains a `<library>` tag, which indicates the start of an XML element node named `library` that ends when a matching `<library>` tag is found.

The `<library>` element is the root element of the document and contains two `<book>` element child nodes, each containing a `<title>` and an `<author>` node.

The `<title>` and `<author>` elements each contain text data.

**XML Tree Architecture**

The JADE XML Tree model reads an XML document from a file or string and builds a tree of objects representing that document.

Your applications can then navigate the tree, extract information from the document, manipulate the tree, and write it into an XML file or string, as shown in the following diagram.

![Diagram showing the JADE XML Tree model](image)

The following diagram shows the JADE XML Tree class hierarchy.

![Class hierarchy diagram](image)

The classes in the JADE XML Tree model are listed in the following table.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JadeXMLAttribute</td>
<td>Defines an XML attribute. Attributes are name value pairs associated with elements.</td>
</tr>
<tr>
<td>JadeXMLCDATA</td>
<td>Defines an XML DATA section. DATA sections are used to escape blocks of text containing mark-up characters.</td>
</tr>
<tr>
<td>JadeXMLCharacterData</td>
<td>Abstract superclass for character data nodes.</td>
</tr>
</tbody>
</table>
### Class | Description
--- | ---
JadeXMLComment | Defines the content of a comment.
JadeXMLDocument | Represents the entire XML document.
JadeXMLDocumentParser | Represents the interface for parsing XML documents into a tree of objects.
JadeXMLDocumentType | Defines an XML DOCTYPE declaration.
JadeXMLElement | Represents an XML element. Elements are the fundamental units of XML.
JadeXMLNode | Abstract superclass of all nodes in the tree.
JadeXMLProcessingInstruction | Defines an XML processing instruction. Processing instructions tell applications how they should handle a document after it has been parsed.
JadeXMLText | Represents character-based content in the document.

For a summary of the properties and methods defined in these classes, if applicable, see "JADE XML Classes", later in this chapter.

A JADE XML Tree document is structured as follows.

- Each XML document has a single JadeXMLDocument node.
- Each JadeXMLDocument node has a single top-level JadeXMLElement node, called the root element.
- Each JadeXMLElement node has a name and zero or more JadeXMLAttribute nodes.
- Each parent node has zero or more child nodes.

The following diagram shows the primary relationships used to model the tree structure.
The following object diagram shows the tree of nodes that make up the example library XML file.

![Diagram of XML tree nodes](image)

**Note**: In this example, the text data is stored directly in the element object, which improves performance. However, for elements that contain mixed content (that is, both elements and text), the text is always stored in separate child *JadeXMLText* nodes.

---

**Using the XML Tree Model**

The following subsections provide instructions for using the XML Tree model.

- Parsing an XML Tree Document
- Processing Extra Whitespace
- Creating XML Tree Documents
- Retrieving Information from XML Tree Documents
- Traversing XML Documents
- Modifying XML Tree Documents
- Writing XML Tree Documents
- Handling XML Tree Exceptions

**Parsing an XML Tree Document**

To parse an XML document into a document object tree

1. Create a *JadeXMLDocument* object.
2. Call the *JadeXMLDocument* class *parseString* or *parseFile* method.
Note  Objects in the tree are created as transient objects by default because that is generally the way they are used. However, you can create them as shared transient objects (or persistent objects), if required, in which case transaction control is your responsibility.

JadeXMLNode objects created with the `add<node-type>` (rather than the `add<node-object>`) methods are created with the same persistence as the node to which they are added.

The following example parses a simple document string into a tree of transient objects:

```javascript
vars
doc: JadeXMLDocument;
begin
create doc;
doc.parseString('<employee>John Smith</employee>');</nend;
```

**Note**  For an XML string to parse correctly, if an element is prefixed by a namespace, the namespace must be specified. The following line of code results in a parser exception.

```javascript
doc.parseString('<wx:employee>John Smith</wx:employee>');
```

To avoid the exception, change the opening tag to the following.

```xml
<wx:employee xmlns:wx="http://required_namespace">
```

To create a tree of persistent objects, you first need to define your own subclasses of all the JadeXMLNode concrete classes. You use the JadeXMLDocumentParser class to map your classes to the JadeXMLNode classes so that node instances will be created in your user map files. You then call the parseDocumentFile or parseDocumentString method. The following example parses a string into a tree of persistent objects.

```javascript
vars
parser: JadeXMLDocumentParser;
doc: MyDocument;
begin
beginTransaction;
create parser;
parser.setClassMapping(JadeXMLElement, MyElement);
parser.setClassMapping(JadeXMLAttribute, MyAttribute);
parser.setClassMapping(JadeXMLText, MyText);
parser.setClassMapping(JadeXMLComment, MyComment);
parser.setClassMapping(JadeXMLCDATA, MyCDATA);
parser.setClassMapping(JadeXMLProcessingInstruction, MyProcessingInstruction);
parser.setClassMapping(JadeXMLDocumentType, MyDocumentType);
create doc persistent;
parser.parseDocumentString(doc, '<employee>John Smith</employee>');
commitTransaction;
end;
```

### Processing Extra Whitespace

By default, JADE assumes that an XML document contains data and automatically discards extra whitespace between adjacent tags. This optimization significantly improves performance by avoiding the need to create extra text objects.
When writing the document, the document tree is written with automatic indentation by default. If the extra whitespace is significant, you can set the `JadeXMLDocument::keepWhitespace` parameter to `true` to instruct the parser to keep the extra whitespace as text nodes.

When writing the document, you can set the `JadeXMLDocument::indentString` property to the empty string, to write the document without automatic indentation.

**Creating XML Tree Documents**

Methods defined in `JadeXMLNode` subclasses enable you to create new documents from scratch. You can create node objects, add them to an XML document, and set their textual content.

The following example illustrates the use of these methods.

```javascript
create1();
vars
doc : JadeXMLDocument;
library, book, title, author : JadeXMLElement;
begin
  create doc;
  doc.addComment('An example XML document');
  library := doc.addElement('library');
  book := library.addElement('book');
  book.setAttribute('isbn', '0-246-13655-3');
  title := book.addElement('title');
  title.setText('Mystery');
  author := book.addElement('author');
  author.setText('Peter Straub');
  book := library.addElement('book');
  book.setAttribute('isbn', '1-876590-17-3');
  title := book.addElement('title');
  title.setText('False Memory');
  author := book.addElement('author');
  author.setText('Dean Koontz');
  write doc.writeString;
  delete doc;
end;
```

The following example shows the output from the `create1` method in the previous example.

```xml
<?xml version="1.0"?>
<!--An example XML document-->
<library>
  <book isbn="0-246-13655-3">
    <title>Mystery</title>
    <author>Peter Straub</author>
  </book>
  <book isbn="1-876590-17-3">
    <title>False Memory</title>
    <author>Dean Koontz</author>
  </book>
</library>
```
To create a persistent document from scratch, you create instances of your own JadeXMLNode subclasses and then add them to the tree. For example, the following example is similar to the previous example except that it creates a tree of persistent objects.

```javascript
create2();

vars
doc: MyDocument;
comment: MyComment;
attribute: MyAttribute;
library, book, title, author: MyElement;

begin
    beginTransaction;
    create doc persistent;
    create comment persistent;
    doc.addCommentObject(comment, 'An example XML document');
    create library persistent;
    doc.addElementObject(library, 'library');
    create book persistent;
    library.addElementObject(book, 'book');
    create attribute persistent;
    book.addAttributeObject(attribute, 'isbn', '0-246-13655-3');
    create title persistent;
    book.addElementObject(title, 'title');
    title.setText('Mystery');
    create author persistent;
    book.addElementObject(author, 'author');
    author.setText('Peter Straub');
    create isbn and title persistent;
    book.addElementObject(title, 'isbn', '1-876590-17-3');
    create title persistent;
    book.addElementObject(title, 'title');
    title.setText('False Memory');
    create author persistent;
    book.addElementObject(author, 'author');
    author.setText('Dean Koontz');
    write doc.writeToString;
    commitTransaction;
end;
```

Retrieving Information from XML Tree Documents

When you have parsed a document into a JadeXMLDocument object, you can search it to select the parts in which you are interested. If you know the names of specific elements in the tree, you can retrieve them using the JadeXMLDocument and JadeXMLElement classes methods. For example, the JadeXMLDocument::findElementsByTagName method retrieves all elements in the document with a specified name.

As the JadeXMLDocument class getElementsByTagNameNS and getElementsByTagName methods scan sequentially to locate requested elements, they always returned requested elements in document sequence but may be relatively slow.
To improve performance, you can use the `JadeXMLDocument` class `findElementByNameNS`, `findElementByTagName`, `findElementsByNameNS`, and `findElementsByTagName` methods and the `JadeXMLElement` class `findAllElementsByNameNS` and `findAllElementsByTagName` methods to retrieve elements more directly through a collection, using the collection sequence. JADE fully supports the use of a mixture of the document and collection sequence methods to locate the requested elements.

The collection sequence methods provide a performance boost only if a `localName` or `tagName` parameter value is explicitly specified in the calling parameters. If you specify `"*"` in the `localName` or `tagName` parameter, the access method reverts to the functionality and performance of the document sequence methods to locate the requested elements.

The code in the following example uses these methods to search the library document and list all books with a specified author.

```java
listBooks(doc: JadeXMLDocument; authorName: String);
vars
    books : JadeXMLElementArray;
    book : JadeXMLElement;
begin
    create books;
    doc.findElementsByTagName('book', books);
    foreach book in books
        where book.getElementByTagName('author').text = authorName do
            write book.getElementByTagName('title').text;
    endforeach;
    delete books;
end;
```

### Traversing XML Documents

When processing particular information in an XML document, you may often need to navigate a tree from the root to the deepest leaf element in document order.

If you want to traverse (walk through) the complete tree in document order, you can use the `childNodes` property array of the `JadeXMLNode` class, which contains the immediate children of each node.

The following examples read a document and print the names of the elements in that document, indented to show the hierarchy.

```java
listChildren(elt: JadeXMLElement; depth: Integer);
vars
    i : Integer;
    str : String;
    child : JadeXMLNode;
begin
    foreach i in 1 to depth do
        str := str & ' ';
    endforeach;
    write str & elt.tagName;
    foreach child in elt.childNodes where child.isKindOf(JadeXMLElement) do
        listChildren(child.JadeXMLElement, depth + 1);
    endforeach;
end;
```

```java
listElements(fileName: String);
vars
    doc : JadeXMLDocument;
```
begin
  create doc;
  doc.parseFile(fileName);
  listChildren(doc.rootElement, 0);
  delete doc;
end;

The following is output when the listElements method is run on the library1.xml document.

library
  book
    title
    author
  book
    title
    author

As an example of a more-general approach for traversing a tree, you could define the following recursive method of the JadeXMLNode class to traverse a sub-tree of the document.

walk();
vars
  child: JadeXMLNode;
begin
  processNode; // process this node
  foreach child in childNodes do
    child.walk;
  endforeach;
end;

In the above example, the polymorphic Node::processNode method would provide the specific code to process each type of node. To process the whole document, simply call the following.

doc.walk;

Modifying XML Tree Documents

JADE provides methods that change the tree to add, move, copy, and remove nodes. The following example changes the order of the books in the library1.xml document.

reorder1();
vars
  doc : JadeXMLDocument;
  books : JadeXMLElementArray;
begin
  create doc;
  doc.parseFile('library1.xml');
  create books;
  doc.rootElement.getElementsByTagName('book', books);
  books[2].moveBefore(books[1]);
  write doc.writeString;
  delete books;
  delete doc;
end;
The output from the `reorder1` method shown in the previous example is as follows.

```xml
<?xml version="1.0"?>
<!--An example XML document-->
<library>
  <book isbn="1-876590-17-3">
    <title>False Memory</title>
    <author>Dean Koontz</author>
  </book>
  <book isbn="0-246-13655-3">
    <title>Mystery</title>
    <author>Peter Straub</author>
  </book>
</library>
```

### Writing XML Tree Documents

A quick and easy way to write an XML document (or any `JadeXMLNode` object) is by using the `writeToString` and `writeToFile` methods. You can override the default indentation and end-of-line sequence by using the `JadeXMLDocument` class `indentString` and `endOfLine` properties.

The following example parses a simple document string and formats the print output.

```javascript
write1();
vars
doc : JadeXMLDocument;
begin
  create doc;
  doc.indentString := '  ';
  doc.parseString('<name><first>John</first><last>Smith</last></name>');
  write doc.writeString;
  delete doc;
end;
```

The output from the `write1` method shown in the previous example is as follows.

```xml
<?xml version="1.0"?>
<name>
  <first>John</first>
  <last>Smith</last>
</name>
```

### Handling XML Tree Exceptions

An exception is raised when an error is detected while processing an XML document. The exception is an instance of the `JadeXMLException` class.
For well-formed parsing errors, the line number and column number of the error is reported. For example, the following methods check that an XML document file is well-formed.

```java
check(fileName: String);
vars
doc : JadeXMLDocument;
begin
  on JadeXMLException do checkExceptionHandler(exception);
  create doc;
  doc.parseFile(fileName);
  write fileName & ' is well-formed';
  delete doc;
end;

checkExceptionHandler(ex: JadeXMLException): Integer;
begin
  write ex.fileName & ' is not well-formed - ' & ex.extendedErrorText &
  ' at line ' & ex.lineNumber.String & ', column ' &
  ex.columnNumber.String & ': ' & ex.errorItem;
  return Ex_Abort_Action;
end;
```

If the document is not well-formed, an XML exception is raised and the checkExceptionHandler method in the above example is invoked to print the error details.

**JADE XML Parser Model**

The JADE XML Parser model reads an XML document from beginning to end.

As it encounters start-tags, end-tags, text, comments, and so on, it notifies the client application by calling event handler methods defined by the application.

**JADE XML Parser Model Architecture**

The JADE XML Parser model architecture is shown in the following diagram.

In your client application, subclass the `JadeXMLParser` class and implement any required callback methods on your subclass.
The following example shows the callback methods that you can implement in your JadeXMLParser subclass.

Using the XML Parser Model

The following subsections provide instructions for using the XML Parser model.

- Parsing an XML Document
- Handling XML Parser Exceptions

Parsing an XML Document

To parse an XML document

1. In your client application, define a subclass of the JadeXMLParser class.
2. Implement any callback methods that you require; for example, the startElement, endElement, and characters methods.
3. Create an instance of the JadeXMLParser subclass and call the parseFile or parseString method.

The following example parses a document and prints the names of the elements in that document, indented to show the hierarchy.

``` GENERATED_SCRIPT
listElements(fileName: String);
vars
    parser : MyParser;
begin
    create parser;
    parser.parseFile(fileName);
    delete parser;
end;
```

The MyParser class is a subclass of the JadeXMLParser class and it has a single Integer attribute depth that keeps track of the current element indent level.
The **MyParser** class implements the following client callback methods.

```java
startElement(namespaceURI, localName, qualifiedName: String;
   attributeCount: Integer) updating, protected;
vars
   i : Integer;
   str : String;
begin
   foreach i in 1 to depth do
      str := str & ' ';
   endforeach;
   write str & qualifiedName;
   depth := depth + 1;
end;

endElement(namespaceURI: String; localName: String;
   qualifiedName: String) updating, protected;
begin
   depth := depth - 1;
end;
```

As the parser reads the input file, it recognizes the start and end of each element and invokes the `startElement` and `endElement` callback methods in the previous example. The `startElement` method prints the indentation and the name of the current element, and increments the hierarchy depth. The `endElement` method decrements the hierarchy depth. The following is output when the `listElements` method is run on the `library1.xml` document.

```
library
book
   title
   author
book
   title
   author
```

### Handling XML Parser Exceptions

The handling of JADE XML Parser exceptions is similar to that for JADE XML Tree exception handling; that is, a `JadeXMLException` is raised when an error is detected parsing an XML document.

The methods in the following examples check that documents are well-formed. Because a tree of objects is not being created, this code checks large documents significantly faster than the documented example for the JADE XML Tree model, under "Handling XML Tree Exceptions", earlier in this chapter.

```java
check(fileName: String);
vars
   parser : MyParser;
begin
   on JadeXMLException do checkExceptionHandler(exception);
   create parser;
   parser.parseFile(fileName);
   write fileName & ' is well-formed';
   delete parser;
end;

checkExceptionHandler(ex: JadeXMLException): Integer;
begin
   write ex.fileName & ' is not well-formed - ' & ex.extendedErrorText &
```
In this example, the `MyParser` class has been defined as an empty subclass of the `JadeXMLParser` class. You do not need to implement any callback methods. If the document is not well-formed, an XML exception is raised and the `checkExceptionHandler` method in the previous example is invoked to print the error details.

You can also raise an exception from within your callback method to stop the parsing of the rest of the document, if required. For example, the following callback method raises an exception when it detects the specific tag for which it is searching. Your exception handler could then handle the exception to meet your requirements.

```java
startElement(namespaceURI, localName, qualifiedName: String; attributeCount: Integer) updating, protected;
vars
  ex : JadeXMLException;
begin
  if qualifiedName = searchString then
    create ex;
    ex.errorCode := MyErrorCode; // indicate string found
    ex.fileName := fileName;
    ex.lineNumber := lineNumber;
    ex.columnNumber := columnNumber;
    raise ex;
  endif;
end;
```

### JADE XML Classes

This section summarizes the following JADE XML classes and the properties and methods defined in those classes, if applicable.

- `JadeXMLAttribute` Class
- `JadeXMLCDATA` Class
- `JadeXMLCharacterData` Class
- `JadeXMLComment` Class
- `JadeXMLDocument` Class
- `JadeXMLDocumentParser` Class
- `JadeXMLDocumentType` Class
- `JadeXMLElement` Class
- `JadeXMLException` Class
- `JadeXMLNode` Class
- `JadeXMLParser` Class
- `JadeXMLProcessingInstruction` Class
- `JadeXMLText` Class

For details, see Chapter 1 of the *JADE Encyclopaedia of Classes*. 
The JadeXMLAttributeArray, JadeXMLElementArray, and JadeXMLNodeArray classes are transient array-based containers whose members are JadeXMLAttribute, JadeXMLElement, and JadeXMLNode objects, respectively, in an XML document tree.

**JadeXMLAttribute Class**

The JadeXMLAttribute class defines the behavior for attributes of XML elements in a document tree. An attribute has a name, an optional namespace, and a value.

The properties defined in the JadeXMLAttribute class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains the …</th>
</tr>
</thead>
<tbody>
<tr>
<td>element</td>
<td>Owning element of the attribute</td>
</tr>
<tr>
<td>localName</td>
<td>Local name (without prefix) of the attribute</td>
</tr>
<tr>
<td>name</td>
<td>Qualified name (with prefix) of the attribute</td>
</tr>
<tr>
<td>namespaceURI</td>
<td>Namespace URI of the attribute</td>
</tr>
<tr>
<td>value</td>
<td>Value of the attribute</td>
</tr>
</tbody>
</table>

The method defined in the JadeXMLAttribute class is summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>namespacePrefix</td>
<td>Returns the namespace prefix</td>
</tr>
</tbody>
</table>

For details, see "JadeXMLAttribute Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

**JadeXMLCDATA Class**

The JadeXMLCDATA subclass of the JadeXMLCharacterData class represents a CDATA section in an XML document tree. CDATA sections are used to escape blocks of text containing characters that would otherwise be regarded as markup.

**Note:** If you do not want the framework to interpret the XML special characters (that is, <, >, & and ”) for a string, call the String primitive type makeXMLCDATA method, which returns a new string of the receiver prepended with <![CDATA[and appended with ]]>.

**JadeXMLCharacterData Class**

The JadeXMLCharacterData class is the abstract superclass of character-based nodes in an XML document tree; that is, the text, CDATA, and comment nodes. The property defined in the JadeXMLCharacterData class is summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains the …</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>Text value of the node</td>
</tr>
</tbody>
</table>

The JadeXMLCharacterData class provides the JadeXMLCDATA, JadeXMLComment, and JadeXMLText subclasses. For details, see Chapter 1 of the JADE Encyclopaedia of Classes.
JadeXMLComment Class

The JadeXMLComment subclass of the JadeXMLCharacterData class represents the content of a comment in an XML document; that is, all of the characters between the starting ' <!--' and ending '-->'.

JadeXMLDocument Class

The JadeXMLDocument class represents an XML document as a tree of nodes. It defines the owning object of all objects in the tree.

The properties defined in the JadeXMLDocument class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>docType</td>
<td>Document type of the document</td>
</tr>
<tr>
<td>endOfLine</td>
<td>End-of-line separator for output</td>
</tr>
<tr>
<td>indentString</td>
<td>Indentation string for output</td>
</tr>
<tr>
<td>keepWhitespace</td>
<td>Specifies whether extra whitespace is discarded</td>
</tr>
<tr>
<td>outputDeclaration</td>
<td>Specifies whether the XML declaration is output</td>
</tr>
<tr>
<td>rootElement</td>
<td>Root element of the document</td>
</tr>
</tbody>
</table>

The methods defined in the JadeXMLDocument class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addComment</td>
<td>Creates and adds a comment</td>
</tr>
<tr>
<td>addCommentObject</td>
<td>Adds a comment object</td>
</tr>
<tr>
<td>addDocumentType</td>
<td>Creates and adds a document type</td>
</tr>
<tr>
<td>addDocumentTypeObject</td>
<td>Adds a document type object</td>
</tr>
<tr>
<td>addElement</td>
<td>Creates and adds an element</td>
</tr>
<tr>
<td>addElementNS</td>
<td>Creates and adds an element with a namespace URI</td>
</tr>
<tr>
<td>addElementObject</td>
<td>Adds an element object</td>
</tr>
<tr>
<td>addElementObjectNS</td>
<td>Adds an element object with a namespace URI</td>
</tr>
<tr>
<td>addProcessingInstruction</td>
<td>Creates and adds a processing instruction</td>
</tr>
<tr>
<td>addProcessingInstructionObject</td>
<td>Adds a processing instruction object</td>
</tr>
<tr>
<td>findElementByNameNS</td>
<td>Returns an element with the specified namespace URI and local name</td>
</tr>
<tr>
<td>findElementByTagName</td>
<td>Returns an element with the specified tag name</td>
</tr>
<tr>
<td>findElementsByNameNS</td>
<td>Fills an array with all elements in the document with the specified namespace URI and local name</td>
</tr>
<tr>
<td>findElementsByTagName</td>
<td>Fills an array with all elements in the document with the specified tag name</td>
</tr>
<tr>
<td>getElementByTagName</td>
<td>Returns the first element with the specified tag name</td>
</tr>
<tr>
<td>getElementByTagNameNS</td>
<td>Returns the first element with the specified namespace URI and local name</td>
</tr>
<tr>
<td>getElementsByTagName</td>
<td>Fills an array with all elements in the document with the specified tag name</td>
</tr>
</tbody>
</table>
The JadeXMLDocumentParser class is the transient class that represents the interface for parsing XML documents into a tree of objects. The parser reads an XML document and creates a tree of object nodes that are instances of the JadeXMLNode classes or user subclasses.

The JadeXMLDocumentParser class provides a more-flexible method of parsing document trees compared to the parseFile and parseString methods of the JadeXMLDocument class. Because you can set up a mapping of user subclasses to node classes, the JadeXMLDocumentParser class enables you to parse documents into persistent trees that reside in user-specified map files.

The methods defined in the JadeXMLDocumentParser class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>comment</td>
<td>Receives notification of a comment</td>
</tr>
<tr>
<td>parseDocumentFile</td>
<td>Parses an XML document file</td>
</tr>
<tr>
<td>parseDocumentString</td>
<td>Parses an XML document string</td>
</tr>
<tr>
<td>processingInstruction</td>
<td>Receives notification of a processing instruction</td>
</tr>
<tr>
<td>setClassMapping</td>
<td>Sets the mapping of a JadeXMLNode class to a user subclass</td>
</tr>
<tr>
<td>startCDATA</td>
<td>Receives notification of the start of DTD declarations</td>
</tr>
</tbody>
</table>

The JadeXMLDocumentType class represents the document type declaration in an XML document tree. A reference to the document type (if the document has one) is stored in the docType property of the JadeXMLDocument class.
The properties defined in the **JadeXMLDocumentType** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>internalSubset</td>
<td>Internal subset</td>
</tr>
<tr>
<td>name</td>
<td>Name of the Document Type Definition (DTD)</td>
</tr>
<tr>
<td>publicId</td>
<td>Public identifier of the external subset</td>
</tr>
<tr>
<td>systemId</td>
<td>System identifier of the external subset</td>
</tr>
</tbody>
</table>

For details, see "**JadeXMLDocumentType Class**", in Chapter 1 of the *JADE Encyclopaedia of Classes*.

**JadeXMLElement Class**

The **JadeXMLElement** class represents an XML element in a document tree. An element can have attributes, child nodes, and textual content.

The properties defined in the **JadeXMLElement** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>attributes</td>
<td>Array of attributes of the element</td>
</tr>
<tr>
<td>localName</td>
<td>Local name (without the prefix) of the element</td>
</tr>
<tr>
<td>namespaceURI</td>
<td>Namespace URI of the element</td>
</tr>
<tr>
<td>tagName</td>
<td>Qualified name (with the prefix) of the element</td>
</tr>
<tr>
<td>textData</td>
<td>Text data of a text-only element</td>
</tr>
</tbody>
</table>

The methods defined in the **JadeXMLElement** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addAttribute</td>
<td>Creates and adds an attribute</td>
</tr>
<tr>
<td>addAttributeNS</td>
<td>Creates and adds an attribute with a namespace</td>
</tr>
<tr>
<td>addAttributeObject</td>
<td>Adds an attribute object</td>
</tr>
<tr>
<td>addAttributeObjectNS</td>
<td>Adds an attribute object with a namespace</td>
</tr>
<tr>
<td>addCDATA</td>
<td>Creates and adds a CDATA node</td>
</tr>
<tr>
<td>addCDATAObject</td>
<td>Adds a CDATA object</td>
</tr>
<tr>
<td>addComment</td>
<td>Creates and adds a comment</td>
</tr>
<tr>
<td>addCommentObject</td>
<td>Adds a comment object</td>
</tr>
<tr>
<td>addElement</td>
<td>Creates and adds an element</td>
</tr>
<tr>
<td>addElementNS</td>
<td>Creates and adds an element with a namespace</td>
</tr>
<tr>
<td>addElementObject</td>
<td>Adds an element object</td>
</tr>
<tr>
<td>addElementObjectNS</td>
<td>Adds an element object with a namespace</td>
</tr>
<tr>
<td>addProcessingInstruction</td>
<td>Creates and adds a processing instruction</td>
</tr>
<tr>
<td>addProcessingInstructionObject</td>
<td>Adds a processing instruction object</td>
</tr>
</tbody>
</table>
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addText</td>
<td>Creates and adds a text node</td>
</tr>
<tr>
<td>addTextObject</td>
<td>Adds a text object</td>
</tr>
<tr>
<td>findAllElementsByNameNS</td>
<td>Fills an array with all descendant elements with the specified namespace URI and local name</td>
</tr>
<tr>
<td>findAllElementsByTagName</td>
<td>Fills an array with all descendant elements with the specified tag name</td>
</tr>
<tr>
<td>findAllElementsByTagNameNS</td>
<td>Fills an array with all descendant elements with the specified namespace URI and local name</td>
</tr>
<tr>
<td>getAttributeByName</td>
<td>Returns the attribute with the specified name</td>
</tr>
<tr>
<td>getAttributeByNameNS</td>
<td>Returns the attribute with the specified namespace URI and local name</td>
</tr>
<tr>
<td>getElementByTagName</td>
<td>Returns the first immediate child element with the specified tag name</td>
</tr>
<tr>
<td>getElementByTagNameNS</td>
<td>Returns the first immediate child element with the specified namespace URI and local name</td>
</tr>
<tr>
<td>getElementsByTagName</td>
<td>Fills an array with the immediate child elements with the specified tag name</td>
</tr>
<tr>
<td>getElementsByTagNameNS</td>
<td>Fills an array with the immediate child elements with the specified namespace URI and local name</td>
</tr>
<tr>
<td>namespacePrefix</td>
<td>Returns the namespace prefix</td>
</tr>
<tr>
<td>parentElement</td>
<td>Returns the parent element of the element</td>
</tr>
<tr>
<td>setText</td>
<td>Sets the text content of the element</td>
</tr>
<tr>
<td>text</td>
<td>Returns the text content of the element</td>
</tr>
</tbody>
</table>

For details, see "JadeXMLElement Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

**Note** As the `getAllElementsByTagNameNS` and `getAllElementsByTagName` methods scan sequentially to locate all requested elements, they always returned requested elements in document sequence. To improve performance, you can use the `findAllElementsByNameNS` and `findAllElementsByTagName` methods to retrieve elements more directly through a collection, using the collection sequence. JADE fully supports the use of a mixture of the document and collection sequence methods to locate the requested elements.

The collection sequence methods provide a performance boost only if a `localName` or `tagName` parameter value is explicitly specified in the calling parameters. If you specify `***` in the `localName` or `tagName` parameter, the access method reverts to the functionality and performance of the document sequence methods to locate the requested elements.

### JadeXMLException Class

The `JadeXMLException` class is the transient class that defines behavior for exceptions that occur as a result of XML processing. The `Exception` class `errorItem` and `extendedErrorText` properties and the `JadeXMLException` class properties are used to describe the XML processing exception in more detail.
The constants provided by the `JadeXMLElement` class are listed in the following table.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CannotParsePersistent</td>
<td>8910</td>
<td>Parse an XML document into a persistent object tree using the <code>parseFile</code> or <code>parseFile</code> method of the <code>JadeXMLDocument</code> class. This is not allowed. To parse persistent documents, you must use the <code>JadeXMLDocumentParser</code> class and set up a mapping of node.</td>
</tr>
<tr>
<td>DocTypeAlreadyDefined</td>
<td>8903</td>
<td>Add a document type declaration to an XML document and one exists already. An XML document can have only one document type declaration.</td>
</tr>
<tr>
<td>InvalidClassMapping</td>
<td>8909</td>
<td>Set an invalid mapping for a <code>JadeXMLNode</code> class. The mapping is used when instances are created during the parsing of an XML document.</td>
</tr>
<tr>
<td>InvalidHierarchyRequest</td>
<td>8905</td>
<td>Add a node to an XML document at an invalid position; for example, moving an element to before a document or attribute.</td>
</tr>
<tr>
<td>NullNode</td>
<td>8904</td>
<td>Pass a null node reference to an XML processing method and the parameter cannot be null; for example, specifying a null value as the destination position when moving a node in the document tree.</td>
</tr>
<tr>
<td>ParserCreateFailed</td>
<td>8900</td>
<td>Create an instance of the XML parsing engine that cannot be created.</td>
</tr>
<tr>
<td>ParserError</td>
<td>8901</td>
<td>Parse an XML document and an error occurs; for example, the document is not well-formed.</td>
</tr>
<tr>
<td>ParserNodeMismatch</td>
<td>8908</td>
<td>Access an XML parser object on a different node to the one that created the parser; for example, when a parser is opened on a client node and a server method attempts to use the parser, this exception is raised.</td>
</tr>
<tr>
<td>RootElementAlreadyDefined</td>
<td>8902</td>
<td>Add a root (top-level) element to an XML document and one exists already. An XML document must have a single root element.</td>
</tr>
<tr>
<td>StringToUTF8Failed</td>
<td>8906</td>
<td>Parse an XML document that cannot be converted from JADE native format to UTF8 format.</td>
</tr>
<tr>
<td>UTF8ToStringFailed</td>
<td>8907</td>
<td>Parse an XML document that cannot be converted from UTF8 format to JADE native format.</td>
</tr>
</tbody>
</table>

The properties defined in the `JadeXMLElement` class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains the …</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>columnNumber</code></td>
<td>Column number of text where the exception occurred</td>
</tr>
<tr>
<td><code>fileName</code></td>
<td>Name of the file where the exception occurred</td>
</tr>
<tr>
<td><code>lineNumber</code></td>
<td>Line number of text where the exception occurred</td>
</tr>
</tbody>
</table>

For details, see "JadeXMLElement Class", in Chapter 1 of the *JADE Encyclopaedia of Classes*. 
JadeXMLNode Class

The JadeXMLNode class is the abstract superclass of all nodes in an XML document tree. A node has an owning document and it can have child nodes and a parent node. A node can be copied, moved, or removed, and it can have its XML representation output. If you know the names of specific elements in the tree, you can retrieve them using the methods defined in the JadeXMLDocument and JadeXMLElement classes. For example, the JadeXMLDocument::getElementsByTagname method retrieves all elements in the document with a specified name, and the JadeXMLElement::getElementByName method returns the first child element that matches a specified name.

The properties defined in the JadeXMLNode class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains…</th>
</tr>
</thead>
<tbody>
<tr>
<td>childNodes</td>
<td>Array of children of the node</td>
</tr>
<tr>
<td>document</td>
<td>Owning document of the node</td>
</tr>
<tr>
<td>parentNode</td>
<td>Parent of the node</td>
</tr>
</tbody>
</table>

The methods defined in the JadeXMLNode class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>copyAfter</td>
<td>Copies the node and inserts it after the specified node</td>
</tr>
<tr>
<td>copyAsChildOf</td>
<td>Copies the node and inserts it as a child of the specified node</td>
</tr>
<tr>
<td>copyBefore</td>
<td>Copies the node and inserts it before the specified node</td>
</tr>
<tr>
<td>descendsFrom</td>
<td>Determines whether the specified XML node is an ancestor of the receiver JadeXMLNode class</td>
</tr>
<tr>
<td>moveAfter</td>
<td>Moves the node to the position after the specified node</td>
</tr>
<tr>
<td>moveAsChildOf</td>
<td>Moves the node to the position as a child of the specified node</td>
</tr>
<tr>
<td>moveBefore</td>
<td>Moves the node to the position before the specified node</td>
</tr>
<tr>
<td>remove</td>
<td>Removes the node from the XML tree and then deletes the node</td>
</tr>
<tr>
<td>writeToString</td>
<td>Writes the node to a string</td>
</tr>
</tbody>
</table>

For details, see "JadeXMLNode Class", in Chapter 1 of the JADE Encyclopaedia of Classes.

JadeXMLParser Class

The JadeXMLParser class is the abstract transient class that defines behavior for parsing XML documents.

The parser reads an XML document and reports basic document-related events; for example, the start and end of elements and character data.

**Note** The order of events is very important, and mirrors the order of information in the document itself. For example, all contents of a JadeXMLElement (that is, character data, processing instructions, and any sub-elements) appear in order between the startElement event method and the corresponding endElement event method.
The methods defined in the **JadeXMLParser** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>characters</td>
<td>Receives notification of character data</td>
</tr>
<tr>
<td>columnNumber</td>
<td>Returns the column number at which the current document event ends</td>
</tr>
<tr>
<td>comment</td>
<td>Receives notification of a comment</td>
</tr>
<tr>
<td>endCDATA</td>
<td>Receives notification of the end of a CDATA section</td>
</tr>
<tr>
<td>endDTD</td>
<td>Receives notification of the end of DTD declarations</td>
</tr>
<tr>
<td>endElement</td>
<td>Receives notification of the end of an element</td>
</tr>
<tr>
<td>fileName</td>
<td>Returns the file name for the current document event</td>
</tr>
<tr>
<td>getAttribute</td>
<td>Retrieves the attribute with the specified index</td>
</tr>
<tr>
<td>getAttributeValueByName</td>
<td>Retrieves the value of the attribute with the specified qualified name</td>
</tr>
<tr>
<td>getAttributeValueByNameNS</td>
<td>Retrieves the value of the attribute with the specified namespace URI and local name</td>
</tr>
<tr>
<td>lineNumber</td>
<td>Returns the line number at which the current document event ends</td>
</tr>
<tr>
<td>parseFile</td>
<td>Parses the specified XML document file</td>
</tr>
<tr>
<td>parseString</td>
<td>Parses the specified XML document string</td>
</tr>
<tr>
<td>processingInstruction</td>
<td>Receives notification of a processing instruction</td>
</tr>
<tr>
<td>startCDATA</td>
<td>Receives notification of the start of a CDATA section</td>
</tr>
<tr>
<td>startDTD</td>
<td>Receives notification of the start of DTD declarations</td>
</tr>
<tr>
<td>startElement</td>
<td>Receives notification of the beginning of an element</td>
</tr>
</tbody>
</table>

For details, see "JadeXMLParser Class", in Chapter 1 of the *JADE Encyclopaedia of Classes*.

### JadeXMLProcessingInstruction Class

The **JadeXMLProcessingInstruction** class represents an XML processing instruction (that is, an application-specific instruction on how to handle an XML document after the document has been parsed). The properties defined in the **JadeXMLProcessingInstruction** class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains the ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>Content of the processing instruction</td>
</tr>
<tr>
<td>target</td>
<td>Target of the processing instruction</td>
</tr>
</tbody>
</table>

For details, see "JadeXMLProcessingInstruction Class", in Chapter 1 of the *JADE Encyclopaedia of Classes*.

### JadeXMLText Class

The **JadeXMLText** subclass of the **JadeXMLCharacterData** class represents the textual content within an XML document tree.

If there is no markup inside the content of an element, the text may be stored directly in the **JadeXMLElement** class **textData** property of the element rather than as a child text node. This optimization reduces the size of the document tree and improves parsing performance.
Chapter 13  
Object Versioning

This chapter covers the following topics.

- Overview
- Specifying Versioning Options
- Versioning a Schema
  - Components of Browsers in Versioned Schemas
  - Displaying a Composite View of Versioned Objects
- Displaying Versioned Entities in All Schemas
- Comparing Versions
  - Viewing the Display of Changes to a Versioned Method
- Removing Versioning from a Schema
  - Comparing the Current and Latest Version of an Entity
  - Inspecting an Entity in the Latest Schema Version

Overview

*Schema evolution* is the process of changing the specifications or description of a JADE system. *Schema instantiation* is the process of bringing those changes to life and making the set of changes current. (For details about schema instantiation, see Chapter 14, "Database Reorganization").

When you make a structural change to a class that has instances (for example, adding, deleting, or changing a property), the class and other related structures are versioned.
Chapter 13  Object Versioning

**Note** If the class is part of an RPS mapping, it is versioned even if it has no instances. This is necessary for the RPS **Datapump** application to behave correctly.

At this stage, there are two versions of the schema: the *current version* and the *latest version*. JADE uses these versions to reorganize the database, for example.

The current version contains the version of the class before the property was added, and matches the current structure of objects in the database. The latest version contains the version of the class after the property was added, although this ‘pending’ change has not yet been brought to life. The process in which the specification of a system changes is called **schema evolution**.

To bring to life the changes in the latest version, they must be made part of the current version. This process is called **schema instantiation**, which includes reorganizing the database. Importantly, much of this process can be carried out online. You need only to shut the system down for the final transition step.

When a JADE system is versioned, you see two versions of a schema where a change has been made; that is, the current schema context and the latest schema context. You can work in the current schema context and make changes to the current version of the system, or you can work in the latest schema context and make changes to the latest version. Changes made in the current schema context may change the behavior of any running application. In addition, not all changes made in the current schema context may be visible in the latest schema context.

For a number of common tasks, object versioning improves the performance of both the versioning and reorganization processes, resulting in faster development. Multiple schema contexts provide snapshots (that is, a version) of a schema specification, enabling you to work in multiple schema versions at the same time. The benefits of object versioning are:

- Enhanced source control capabilities
- Faster development
- Making structural changes to the application while it is still running
- Ability to work in multiple schema versions at the same time

Running an application from the JADE development environment by clicking the **Run Application** toolbar button and methods executed from the **JadeScript** class and a Workspace use the current version of objects.

The following is a summary of the main visible components of versioning in the JADE development environment.

- The Schema Browser can show multiple instances of a specific schema. If a schema is versioned, both the current and latest schema context are displayed in the Schema Browser.

  Although the latest version of a schema is displayed with a green background highlight by default, you can define your preferred version highlight color. You must select the appropriate schema version whose development environment forms you want to open.

- The Class Browser includes **Latest** in its title if the browser is opened against the latest schema context. The Class Browser always displays only one instance of each class.

- A composite view is available for the main source windows, which enables you to view additions, changes, and deletions to class constants, properties, and methods for a versioned class.

  A composite view is available from the current or the latest schema context. If you are viewing a changed class constant, property, or method in a composite view, the browser displays two entries (one for the current and one for the latest version).

- The **Cashmere** JADE skin is used by default to decorate any source window that is opened against the latest (not committed) version of a schema.
Tip  We recommend that you use a JADE skin to indicate versioning, as it provides strong visual feedback of the versioned state of your schema.

If required, remove or change the skin for the current version from the Window sheet of the User Preferences dialog and for the latest version from the Miscellaneous sheet of the User Preferences dialog.

- Icons that are displayed for a versioned element include a small superimposed arrow that indicates the version of the element.

  A white arrow pointing towards the left indicates the current version of the element and a green arrow pointing towards the right indicates the latest version of the element. (If the property or method is not versioned, no arrow is displayed in the icon.)

For an example of a composite view of browsers in a versioned schema, see "Components of Browsers in Versioned Schemas", later in this chapter.

### Specifying Versioning Options

The Versioning Options group box on the Miscellaneous sheet of the JADE Installation Preferences dialog or the JADE development environment Preferences dialog enables you to specify versioning options for all users signing on to the JADE development environment or to your own work sessions, respectively.

You can override the default options to display:

- A warning dialog advising every user signed on to the JADE development environment who has a window open against a newly-versioned schema that the specified schema is now versioned. By default, a warning is not displayed when schemas are versioned.

- A warning dialog when an attempt is made to change source code in a versioned schema. By default, a warning is not displayed when source code is changed.

- A JADE skin to decorate opened source windows in the latest (that is, uncommitted) version of schemas, to provide a distinction between current version and latest version windows.

  By default, the Cashmere skin is used for the latest (uncommitted) version and the Lincoln skin for the current (committed) version in development environment windows. (For details about skins, see "Using JADE Skins in Your Runtime Applications", in Chapter 2 of the JADE Runtime Application Guide.)

- The composite view in new windows opened in versioned schemas rather than the selected version only. By default, the selected version view is displayed. Composite views show all versions of an object (for example, a class, property, or method in one window), with icons indicating the versions.

  Note  You can toggle the display of the composite view and a selected version only view, by selecting the Show Composite View command from the View menu. For details, see "Displaying a Composite View of Versioned Objects", later in this chapter.

For details about specifying versioning preferences that apply to all windows in JADE development environment work sessions for all new users or for your own work sessions, see "Maintaining Miscellaneous Options", in Chapter 2 of the JADE Development Environment User's Guide, respectively.
Versioning a Schema

To version a schema

1. Perform one of the following actions.
   - In the Schema Browser, select the schema that you want to version and then select the Version command from the Schema menu. This is displayed as the Unversion command when the schema is currently versioned. For details about toggling schema versioning, see "Removing Versioning from a Schema".
   - Changing a property may cause the schema to be versioned and methods that reference this property to be versioned and recompiled. If those methods are in different schemas, those schemas are also versioned.

   **Note** For details about the other changes made to classes for which persistent instances exist that may version your schema and require reorganization of the database, see "Changes Requiring a Reorganization", in Chapter 14.

When the schema is already versioned (as a result of a property change, for example), packages exported by a versioned schema may affect schemas importing that package, so the importing schemas may also be versioned and changing a method in an interface mapped to a class in a versioned schema versions the schema implementing the mapped method when the interface and class methods are compiled.

2. A Warning box is then displayed, advising you that the operation versions the specified schema and prompting you to click the Yes button to continue with the schema versioning operation or the No button to abandon the operation.

Another (latest and uncommitted) version of that schema is then created, with the newly created latest version highlighted with a light green background, as shown in the following image.

![Schema Browser with versioned schemas](image)

The Schema Browser always shows versioned schemas. The latest schema, class, and interface versions are displayed with a default background color of light green in the respective browsers.

If the Warn when schemas are versioned check box on the Miscellaneous sheet of the Preferences dialog is checked, a Warning box is displayed to all users signed on to the JADE development environment who have a window open against the newly-versioned schema so that they can continue working in the current schema version or change to the latest (uncommitted) version.

Reorganizing all classes in a versioned schema updates the current (committed) schema version and removes the latest version.
Until no further classes in the versioned schema require reorganization and the current schema is updated, the **Remove** command in the Schema menu is disabled in latest version of the schema. However, if you remove the current version of a schema, the latest version is also removed, even though classes require reorganization.

**Tip** If you intend to display a browser for both the current and the latest version context (that is, a composite view), you can select a JADE skin to decorate opened source windows in the latest version, to provide a distinction between current version and latest version windows.

By default, the **Cashmere** skin is used for the latest (uncommitted) version in windows and the **Lincoln** skin to for current (committed) version. (For details, see "Maintaining Miscellaneous Options", in Chapter 2 of the JADE Development Environment User’s Guide.)

**Components of Browsers in Versioned Schemas**

You can toggle the display of the composite view and a selected version only view, by selecting the **Show Composite View** command from the View menu. For details, see "Displaying a Composite View of Versioned Objects", in the following section.

In a browser of a versioned schema:

- Method, property, and constant elements display an icon appropriate to their access rights (that is, a lock for protected elements, glasses for read-only elements, and a person for public elements). These symbols and the View menu commands that you can select to hide or display these icons in the current browser are listed in the following table.

<table>
<thead>
<tr>
<th>No Version</th>
<th>Description</th>
<th>Command to Toggle Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>🔒</td>
<td>Property, method, or constant is public</td>
<td>Show Public</td>
</tr>
<tr>
<td>🔐</td>
<td>Property or method is protected</td>
<td>Show Protected</td>
</tr>
<tr>
<td>_READONLY</td>
<td>Property is read-only</td>
<td>Show Read Only</td>
</tr>
<tr>
<td>_secure</td>
<td>Property is a key</td>
<td>Not applicable</td>
</tr>
<tr>
<td>✗</td>
<td>Method contains one or more compilation errors</td>
<td>Toggling not applicable to method error display</td>
</tr>
<tr>
<td>✓</td>
<td>Entity in the Methods List is a condition</td>
<td>Conditions</td>
</tr>
</tbody>
</table>

If a method, property, or constant is versioned, the left arrow or right arrow is superimposed onto the icon to indicate the version of that element; that is, a left arrow indicates the current version and the right arrow indicates the latest (uncommitted) version.

If you have a browser open against the current version of the schema, all icons are unadorned or they have a superimposed left arrow that indicates that the method or property is versioned.

If you have a browser open against the latest version of the schema, all icons are unadorned or they have a superimposed right arrow that indicates that the method or property is versioned.

The current version icon indicates the element with which the system is currently running and the latest version icon indicates the latest (uncommitted) version of that element.
Browsers can display a composite view, which shows:

- Any versioned class is highlighted with the user-defined versioned background color, which is light green by default. The same background color is applied to the latest version of any method, property, or class constant.
- Both versions of any versioned method, property, or class constant.
- In the current version context, methods, properties, or constants that have been added to the latest version context.

To provide additional feedback in the current version context, foreground colors in the browsers indicate one of three versioning states, as follows.

- Changed color (which defaults to orange) indicates that a property or class constant has changed between the current and latest version.
- Added color (which defaults to pink) indicates a new method, property, or constant added to current context.
- Removed color (which defaults to gray) indicates a method, property, or class constant that exists in the current schema context and that has been removed from the latest schema context. This element is removed from the current schema when the class is next reorganized.

You can change the version display foreground and background colors, by using the Window sheet of the Preferences dialog, accessed from the Preferences command in the Options menu of browse windows. (For details, see "Maintaining Window Options", in Chapter 2 of the JADE Development Environment User’s Guide.)

### Displaying a Composite View of Versioned Objects

By default, only the selected version of a window is displayed in versioned schemas (that is, the current version or the latest version. However, you can show all versions of an object (for example, a class, property, or method) in one window, with icons indicating the versions, and you can toggle the display of the composite view and a selected version-only view. See also "Components of Browsers in Versioned Schemas", in the previous section.

» To display a composite view, perform one of the following actions

- Select the Show Composite View command from the View menu.
- Select the Show composite version view by default value from the list in the Default viewing preference for new windows list box in the Versioning Options group box on the Miscellaneous sheet of the Preferences dialog.

When a composite version view is displayed, a check mark is displayed to the left of the command in the View menu.

» To display a selected version view of windows, perform one of the following actions

- Select the Show Composite View command from the View menu.
- Select the Show selected version view by default value from the list in the Default viewing preference for new windows list box in the Versioning Options group box on the Miscellaneous sheet of the Preferences dialog. For details, see "Maintaining Window Options", in Chapter 2 of the JADE Development Environment User’s Guide.
Displaying Versioned Entities in All Schemas

You can display and inspect the current and latest version of all entities that are versioned in all schemas in your JADE database.

To display all versioned entities in schemas

- Select the Display Version Info command from the Browse menu.

The Display Version Info window is then displayed.

For details about filtering the versioned entities that are displayed and inspecting an entities, see "Comparing the Current and Latest Version of an Entity" and "Inspecting an Entity in the Latest Schema Version", under "Removing Versioning from a Schema", later in this chapter.

Comparing Versions

Use the Compare Versions command from the Methods menu in the Class Browser, Primitive Types Browser, or Interface Browser to compare the current and latest versions of a method source.

Note: Method source versions can be compared only when versioning has been set for the schema and when there have been changes to the current method.

You can change your default source comparison options, if required, by using the Compare Sources group box controls of the Source Management sheet from the Options menu Preferences command. For example, you can select the comparison of changes only or that white space is to be compared.

To compare the current and latest versions of a method source

1. In the Methods List of the Class Browser, Primitive Types Browser, or Interface Browser, select the versioned method whose sources you want to compare.

2. Select the Compare Versions command from the Methods menu. (This command is disabled if the current version of the method is the only version or if versioning of the schema is not set.)

The Compare Sources window is then displayed. For more details, see "Comparing a Method Source", in Chapter 3 of the JADE Development Environment User's Guide.

Viewing the Display of Changes to a Versioned Method

You can view only the changes made to the source of a versioned method, by selecting the Changes Only option button in the View Options group box of the Source Management sheet, accessed from the Options menu Preferences command.

The Compare Sources window is then displayed. In general, source modifications are displayed in the following format.

current-source indicator latest-source
The locators a, c, or d represent the type of add, change, or delete modification, respectively, that was made for a specific line number. For example:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2d1</td>
<td>Line 2 in current-source has been deleted</td>
</tr>
<tr>
<td>4c3</td>
<td>Line 4 in current-source has been changed to line 3 in latest-source</td>
</tr>
<tr>
<td>6a7</td>
<td>Line 7 in latest-source has been added after line 6 in current-source</td>
</tr>
<tr>
<td>9a11</td>
<td>Line 11 in latest-source has been added after line 9 in current-source</td>
</tr>
</tbody>
</table>

The Context list box in the View Options group box of the Method sheet of the Preferences dialog enables you to specify the number of lines that are displayed on either side of the changed lines when changes only are selected, if required. (By default, lines on either side of changed lines are not displayed.)

## Removing Versioning from a Schema

You can remove versioning from a schema without performing a schema reorganization, if required (for example, if you did not mean to version the schema or if a reorganization fails).

**Note** You cannot remove versioning after performing a schema load into the latest version of a schema with only structural changes. This is not permitted because any method that was recompiled has references to the new versions of classes and properties.

### To remove versioning from a schema

1. In the Schema Browser of the current (committed) schema, select the schema whose versioning you want to remove and then select the Unversion command from the Schema menu. (This is displayed as the Version command when the schema is not currently versioned. For details about toggling schema versioning, see "Versioning a Schema", earlier in this chapter.)

**Note** If removal of versioning is attempted when a reorganization is in progress (regardless of whether the reorganization progress dialog is displayed), the load fails. For details about reorganizing schemas, see "Reorganizing Your Schema", in Chapter 3 of the JADE Development Environment User’s Guide.
The Unversion Schemas dialog, shown in the following image, is then displayed.

2. To view details of the changes in the schema version, click the **Details** button. The Display Version Info window, shown in the following image, is then displayed.
The number of changes made to the latest schema version of all schemas in your JADE database is displayed in parentheses in the title bar of the dialog.

To view schema version information:

a. In the Date From text box in the Selection Criteria group box, specify the starting date from which schema version changes are to be displayed; for example, 03/07/2005 or 03JULY2005. Enter the date in the ddMMMyyyy format. If you do not specify a value in this text box, the search for changed entities starts at the earliest timestamp; that is, the first date on which an entity was changed in the latest schema version.

b. In the To text box in the Selection Criteria group box, specify the date up to which schema version changes are to be displayed; for example, 31/07/2005 or 31JULY2005. Enter the date in the ddMMMyyyy format. If you do not specify a value in this text box, the search for changed entities ends on the current date.

c. In the Entity Name text box of the Selection Criteria group box, specify the name of a specific entity if you want to display schema version changes to that entity only; for example, the TestSchema::JadeScript::test1 method. If you do not enter a value in this text box, all entities that match your other selection criteria are displayed.

d. In the Type drop-down list box in the Selection Criteria group box, specify the meta schema type whose patch version changes you want to display; for example, ExternalMethod, Class, JadeMethod, Interface, or PrimAttribute. If you do not select a type, schema version changes to all meta schema types are displayed.

e. Click the Reset button to clear all controls in the Selection Criteria group box; that is, specify all changes for the latest version of all schemas.

f. Click the Refresh button to initiate the schema version display. Alternatively, click the Close button to abandon your selections.

The display is then initiated. Each schema version change that satisfies any specified search criteria is then displayed in the lower portion of the window when the change is made.

**Tips** To change from the default descending time sort order, click the list header for the element that you want sorted alphabetically or numerically; for example, click the Entity Name list header to list all entities in alphabetical order.

To display and compare method source changes, double-click the appropriate method entity in the display window. For details, see "Comparing the Current and Latest Version of an Entity", in the following section. For details about inspecting a change in the latest schema version, see "Inspecting an Entity in the Latest Schema Version", later in this chapter.

You can resize the Display Version Info window vertically, to enable you to view more information, if required. (You cannot resize this window horizontally, however.)

3. To remove versioning from the schema, including all changes made to the schema, click the Unversion button.

**Caution** This action cannot be undone.

The latest version of the schema and all changes in that schema are then removed and only the current version of schema is displayed in the Schema Browser.
If you add a subschema to a schema that is currently versioned (through the JADE development environment or from a schema load), you cannot remove versioning from the versioned schema. An error message box, advising that the unversioning failed because of subschema dependencies, is displayed when you click the **Unversion** button on the Unversion Schemas dialog.

**Comparing the Current and Latest Version of an Entity**

To compare the current (committed) version of an entity with the latest version, double-click the appropriate entity in the Display Version Info window table.

1. **To compare two versions of a schema entity**
   1. Double-click the entity in the Display Version Info window table whose versions you want to compare. The Version Info window is then displayed.
      
      The **Current** pane at the left of the window displays in green all information in the committed version of that entity that has changed in the latest version. The **Latest** pane at the right of the window displays in green the changed information for the latest version of the entity.
   2. To display all details about the entity, double-click the entity in the **Latest** pane.
The Version Info window is then updated with all information for that entity, as shown in the following image. Green text indicates changes between the current and latest versions.

![Version Info Image]

As the Version Info window is read-only, no source code can be changed.

3. To view the next change in the displayed entity, click the **Next** button. (The **Next** button is disabled if the no other changes to that entity were made in the latest version.)

   The next line entity value that was changed in the latest schema version is then highlighted in both versions of the entity.

4. To view the previous change in the displayed entity, click the **Previous** button. (The **Previous** button is disabled if the first changed line is currently highlighted.)

   The previous value in the entity that was changed in the latest version is then highlighted in both versions of the entity.

5. To return to the Display Version Info window when you have compared both versions of the entity, click the **Cancel** button.
Inspecting an Entity in the Latest Schema Version

To inspect details of the latest version of a schema entity

1. Right-click the entity in the Display Version Info window table whose latest version information you want to view.

2. Select the **Inspect** command from the popup menu.

   The Schema Inspector form for that entity is then displayed. The title bar of the form states the entity whose latest version you are inspecting. The list at the left contains an entry for each property defined in or inherited by that entity, or line of code in a method.

3. To examine a component, select the component in the left-hand list. The details for that component are then displayed at the right of the Schema Inspector form.

**Note**  If the current property is a reference or a collection, you can double-click on it to open a new Inspector form for that property. This enables you to "drill-down" through your references and collections.

You can inspect the edition or time an object was created, by clicking **…edition** or **…creationTime**, displayed beneath **...Object...** in the Object List at the left of the Inspector form.

For details, see "Using the Inspector Form", in Chapter 6 of the *JADE Development Environment User's Guide*.

Using the Options Menu

To specify the font used to display text in the Schema Inspector form

- Select the **Font** command from the Options menu if you want to change the default font from Tahoma, regular, 8.25 points.

The common Font dialog is then displayed, to enable you to make your font selections.

When you have selected your required font options, focus is then returned to the Schema Inspector form.

Using the Help Menu

Use the commands in the Schema Inspector window Help menu to access the standard Common User Access (CUA) help options. These commands are listed in the following subsections.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>Opens the Schema Inspector online help, with all topics available in the online help file displayed down the left side of the help window in Adobe Reader.</td>
</tr>
<tr>
<td>About</td>
<td>Accesses information about the release of the JADE Schema Inspector. (For details, see &quot;About Command&quot; under &quot;Help&quot;, in the <em>JADE Schema Inspector Utility User’s Guide</em>. )</td>
</tr>
</tbody>
</table>

Use the functions available in JADE online help to find the required topics. For details, see "JADE HTML5 Online Help" or "JADE Product Information Library in Portable Document Format", in Chapter 2 of the *JADE Development Environment User’s Guide*. 
Chapter 14  Database Reorganization

This chapter contains the following topics.

- Overview
- Changes Requiring a Reorganization
- Reorganization Options
  - Allowing Updates
  - Initiating Transition
  - Replayable Reorganizations
- Phases of a Reorganization
  - Object Conversion
  - Transition
  - Relationship Maintenance
  - Commit Reorganization
- Reorganization Log Files
- Correcting Data if a Reorganization Fails
  - Restarting Reorganizations
  - Aborting Reorganizations
- Reorganization Restrictions
  - Features Restricted by the JADE Development Environment
  - Unsupported Features
  - Disk Space Requirements
- SDS Reorganization Considerations

Overview

If a change is made to a schema entity that results in the schema being versioned, a reorganization is required to bring the changes into effect.

Structural changes to classes with persistent instances result in these instances being updated when the database is reorganized. Structural changes affecting persistent class instances include:

- Adding a property
- Deleting a property
- Changing the type or length of a property
- Changing the definition of a dictionary or collection (ascending keys, uppercase, duplicates, and so on)
Changing the membership of a collection

Changing tuning parameters for a collection

Changing the collection type

Adding a constraint or changing a condition

Adding or changing an inverse definition

Changing an RPS mapping (which versions the schema)

Using JCF commands to remap classes and move class instance

**Notes**  A class may not be marked for reorganization when you add a reference that is an exclusive collection or a virtual property.

Changing a property that is a key causes all collections of which that property is a key to be reorganized.

Properties are matched by name within the hierarchy of the current class. If a property is deleted and a new property added with the same name, the reorganization assumes that the original property was modified. If the type of the property is unchanged, the value is not altered by the reorganization (although the reorganization must still be initiated). Changes to properties in a class require a reorganization of all subclasses of that class that have instances.

If a class with a JadeBytes attribute is modified in a way that does not affect the Unstructured Data Resource (UDR) file or files (for example, by adding an Integer property), the resulting reorganization skips the UDR file or files.

For details about performing a reorganization, see "Performing a Reorganization", in Chapter 3 of the JADE Development Environment User’s Guide. For details about RPS changes that cause a reorganization, see "Schema Instantiation", in Chapter 2 of the JADE Synchronized Database Service (SDS) Administration Guide.

### Changes Requiring a Reorganization

This section covers the changes that require database reorganization.

**Adding a Property**

When adding a new property, the value is initialized to null, by default.

**Deleting a Property**

When deleting a property, the old value is removed.

**Changing a Property**

Modifying a property includes changing the size, type, or attributes of the property, as follows.

- Changing the size of a String, StringUtf8, or Binary primitive type property.
  
  If the length is decreased, the new value is truncated, if necessary.

- Changing the scale factor or precision of a Decimal primitive type property.
  
  If the scale factor is decreased, the value is truncated, if necessary. If the precision is decreased, the reorganization will fail if any value exceeds the new precision.
Changing the primitive type. The primitive type conversion rules applied for all primitive type to primitive type conversions are those of the JADE language. For details, see "Converting Primitive Types", in Chapter 1.

For the following undefined conversions, the new value is null.

- **Point** primitive type to a non-**Point** primitive type.
- Any primitive type to or from a reference.
- A collection to or from a non-collection (Arrays, Dictionaries, and so on).

Changing the type of a reference (excluding interface references and collections).

Changing the type of a reference to a superclass of the current class does not require a reorganization.

Changing the type of a reference to a subclass of the current class requires a reorganization. All instances of the reference must be null or must refer to an instance of the new class or a subclass of that class.

Changing the type of a reference to a completely different class requires a reorganization. All instances of the reference must be null.

Changing the type of an interface reference.

Changing the type of an interface reference to an interface that is extended by the current interface does not require a reorganization.

Changing the type of an interface reference to an interface that extends the current interface requires a reorganization. All instances of the reference must be null or must refer to an instance that implements the interface or an instance that extends that interface.

Changing the type of an interface reference to a completely different interface requires a reorganization. All instances of the reference must be null.

Changing the type of a collection reference (excluding collections of interfaces).

Changing the type of a collection reference to a superclass of the current collection does not require a reorganization as long as the keys remain valid.

Changing the type of a collection reference to a subclass of the current collection class requires a reorganization. All instances of the collection reference must be empty or must refer to an instance of the new collection class or a subclass of that class.

Changing the type of a collection reference to a completely different collection class requires a reorganization. All instances of the reference must be empty.

Changing the type of a collection of interfaces.

Changing the type of a collection of interfaces to a collection of interfaces that are extended by the current interface does not require a reorganization as long as the keys remain valid.

Changing the type of a collection of interfaces that extend the current interface requires a reorganization. All instances of the collection must be empty or must refer to an instance that implements the interface or an instance that extends that interface.

Changing the type of a collection of interfaces to a completely different collection of interfaces requires a reorganization. All instances of the reference must be empty.

Moving an exclusive collection to a superclass or a subclass requires a reorganization.

Moving a non-embedded **String**, **StringUtf8**, or **Binary** to a superclass or a subclass requires a reorganization.
A non-embedded StringUtf8 or Binary has a length greater than 540. A non-embedded String has a length greater than 539.

- Changing the property attributes. Changing a property to or from virtual from or to non-virtual is handled as though the property were added or deleted, and requires reorganization.
- Changing a property that is the key of a dictionary will result in all instances of the dictionary requiring a reorganization. For details, see "Changing Membership of a Collection", later in this chapter.

Changing Dictionary or Collection Definitions

Changing the definition of a dictionary requires all instances of that dictionary, both exclusive and shared, to be reorganized.

Dictionary changes requiring reorganization include:

- Adding or removing keys
- Changing ascending or descending order
- Changing case-insensitive or case-sensitive
- Duplicates allowed or not allowed

If you add new keys to an ExtKeyDictionary class, they are initialized to null.

If you decrease the length of a String or Binary key, all key values must be less than or equal to the new key length. Instances of String or Binary keys that are longer than a decreased key length cause the reorganization process to fail and an exception to be raised.

**Note** Changing the type of keys of an ExtKeyDictionary class is not supported and is enforced by JADE.

Changing Membership of a Collection

This section covers modifying collection membership.

- Changing the membership of a collection (excluding collections of interfaces)

  Changing the membership of a collection reference to a superclass of the current collection does not require a reorganization as long as the keys remain valid.

  Changing the membership of a collection reference to a subclass of the current membership class requires a reorganization. All instances of the collection must be empty or must refer to an instance of the new membership class or a subclass of that class.

  Changing the membership of a collection reference to a completely different membership class requires a reorganization. All instances of the collection must be empty.

- Changing the membership of a collection of interfaces

  Changing the membership of a collection of interfaces to an interface that is extended by the current interface does not require a reorganization as long as the keys remain valid.

  Changing the membership of a collection of interfaces to an interface that extend the current interface requires a reorganization. All instances of the collection must be empty or contain instances that implements the interface or instances that extends that interface.

  Changing the membership of a collection of interfaces to a completely different interface requires a reorganization. All instances of the collection must be empty.
Changing Tuning Parameters For a Collection

Modifying the following parameters for tuning the storage of a collection class with instances requires a reorganization.

- Expected Population
- Entries Per Block

Changing the Collection Type

The type of a property can be changed from one type of collection to any other type of collection (external key dictionary, member key dictionary, array, or set). These changes require that any existing exclusive collections be converted to the relevant format.

Changing a collection from exclusive to shared results in the creation of a new instance of the shared collection and the contents of the exclusive collection are copied to this new collection. The reference in the parent object is set to this new collection.

As changing a collection from shared to exclusive does not automatically copy the contents to the new exclusive collection, you must do this after the reorganization has completed.

Changing Conditions

If a constraint is added to an inverse that currently has no constraint, a reorganization of the classes involved will be required if the classes have instances. Similarly, if a constraint on an inverse is changed (by altering the underlying condition or using a different condition), the classes involved in the inverse will require reorganization.

If the constraint is new or has changed, the condition will be evaluated (or re-evaluated, if the constraint has changed) for each instance, to ensure that each inverse is established at least once.

If a constraint is added to an inverse and existing inverses are defined on the property, the constraint will also apply to these inverses. These inverses will be validated by evaluating the condition for each instance, to ensure that the inverse is established at least once.

Adding or Changing Inverse Definitions

If an inverse definition is added to a reference that has existing instances, a reorganization is required, to ensure that the inverse is valid. If one or both properties have a value, the reorganization process will attempt to establish the inverse. If an object that does not exist is referenced, a warning message is logged and the reference is set to null.

If both properties in the inverse are defined but do not correspond to each other, an error message is logged.

Notes

If the maintenance of an array is changed from manual to automatic by the definition of an inverse reference, the resulting reorganization populates the array. As no checking for duplicates is performed, any entries that were added manually are duplicated by the reorganization.

When an inverse reference is added to a property that has one or more existing inverse references, the new inverse reference is populated from the manual property.

When reorganization populates an array, the order in which objects are added is not defined. If the order is important, you must manually sort the array after the reorganization has completed.
Reorganizing Inverses when Both Properties Are New

New properties are initialized to null by default, so inverses added to new properties will always be empty.

Reorganizing Inverses when One Property Is New

When reorganizing inverses in which one property is new, the following actions occur.

- The inverses are populated with non-null references, by reading all instances of the referencing class.
- The new property is populated from the existing property.
- If the existing property is a collection, each collection member is processed in turn.
- If the existing property references a non-existent object, the property is set to null.

Reorganizing Inverses when Both Properties Exist

When reorganizing inverses when both properties exist, the following actions occur.

- All instances of both classes are processed.
- If the existing property is a collection, each collection member is processed in turn.
- All instances of the properties in both classes are verified.
- If a non-existent object is referenced, the property is set to null. If the property is a collection, an error message is reported.
- If the inverse of an object being referenced is null, a valid relationship is defined by setting the value of the inverse to the reference.
  - If the inverse is a collection that does not contain the reference, the reference is added to the collection.
  - If the inverse of an object being referenced refers to another instance, the inverse is invalid, and an error message is reported.
  - If a manual to automatic inverse relationship is added to a property and one or more relationships are already defined to the same class, the new inverse is populated from the manual property.

For all other reference update modes, the new inverse is not populated and the application must manually define the inverses that are maintained.

Changing an RPS Mapping

Any change to an RPS mapping, whether to a table or column included in the mapping or to one of the mapping options, causes the schema to be versioned. This in turn requires the schema to be reorganized. For more details, see "Versioning a Schema", in Chapter 13.

When no reorganization is required after a .scm file load, the .ddb or .ddx file can be loaded (as normal) before the reorganization is performed.

If control classes require reorganization after the .scm file load and before the .ddb or .ddx file load, load the .ddbx file (RPS mapping information) before the reorganization, to keep the RPS mappings up-to-date with the schema changes. After the reorganization, the .ddb or .ddx file is loaded as normal, to complete the changes. (When extracting a schema with an RPS mapping and the DDX format style is selected, a separate .ddbx file is produced as for a DDB extract but the file contents are in XML format.)
Using JCF Commands to Remap Classes and Move Class Instances

Use the class remap and move class instances functionality in the batch JADE Schema Load utility (jadloadb), or the JadeSchemaLoader application in jadclient.jade, or the Application class startApplicationWithParameter method commandFile parameter to change class to database file mappings (class maps) and have existing instances moved to match the altered specification in a deployed system.

The operation that moves objects from one database file to another is a special form of database reorganization. Standard operational procedures and precautions recommended when performing a database reorganization therefore apply.

**Note** When changing the database file to which a class is mapped, the database file can be defined anywhere within the scope of the current schema or superschema hierarchy; that is, in the schema that defines the class or in any user superschema.

The main characteristics and benefits of this functionality are:

- Moving objects from one database file to another is implemented in the database engine as a special type of file reorganization that is similar to a file compaction.
- A file split operation in which instances of multiple classes are moved to a single destination file is processed in a single pass of the source file.
- Multiple split operations can be executed in parallel, by using a user-specified number of worker threads.
- Move object operations are replayable on SDS secondaries and roll-forward recovery, with minimal auditing overhead.
- You can specify your own class remap definition and execute the move operation, by using the following JADE command file (JCF) commands in the commandFile parameter.

For details about the syntax and command examples, see "Loading Schemas in Batch Mode using jadloadb", in the Jade Schema Load User's Guide.

- Create DbFile
- Delete DbFile
- Remap Class
- MoveInstances

Use these commands in a JADE command file to change the database file mappings for classes with existing instances in a deployed system.

Although you can execute the MoveInstances process online, the files involved are write-locked and not available for updating during the move operation.

The specification of a remap and move instances task entails the creation of a command file comprising specification commands and a final command that triggers the execution of the MoveInstances process.

**Moving Class Instances**

When the MoveInstances command is processed in the JCF commandFile parameter in the batch Schema Load utility (jadloadb), the pending class remap specifications are analyzed and grouped in order to achieve the required splits with a minimal number of passes of the source file.
A file split operation that entails moving instances of several classes to at most one other target file is processed as a single task in one pass of the source file. You can process multiple file split tasks for different source files concurrently, using the specified number of worker threads.

When classes are remapped from a source file to more than one target file, each file split is processed as a separate task. Move operations are executed in batches or work units comprising one or more file split tasks. More than one work unit is required when the moves from a specified source file cannot be accomplished in a single pass of that file.

Each work unit is executed within a single atomic transaction that commits (or aborts) the reorganization and instantiation of database files as well as associated schema metadata updates.

Information (including errors) about the MoveInstances operation is written to the move.log file in the location specified by the ActivityLogDirectory parameter in the [PersistentDb] section of the JADE initialization file.

### Reorganization Options

Schema reorganization provides you with the following options.

- **Allowing Updates**
- **Initiating Transition**
- **Replayable Reorganizations**

For details, see the following subsections.

#### Allowing Updates

By default, other users can continue development and update the database while the reorganization is in progress; for example, other users can still modify (that is, edit and compile) methods in the development environment). However, you can specify that updates are not allowed during the reorganization. Updates are not allowed when the FastBuildBTreeCollections parameter in the [JadeReorg] section of the JADE initialization file is set to true.

**Notes** Changes that would require a reorganization cannot be made while a reorganization is in progress (for example, adding or removing properties).

Instances of classes being reorganized can be created, updated, or deleted prior to the transition.

If concurrent updates are disallowed, reorganization cannot be initiated if there are any outstanding transactions. If other users are using the JADE database, you must wait until all transactions are complete before you initiate the reorganization. No transactions can be started when a reorganization that disallows concurrent updates is in progress.

If updates are disabled while the reorganization is in progress, the reorganization must initiate the transition. Any error occurring during the reorganization results in the reorganization being aborted; that is, the reorganization cannot be restarted.

#### Initiating Transition

All applications that are using the schemas and classes involved in the reorganization must be closed before the transition can proceed.

If the transition is initiated but cannot proceed because running applications are using schemas or classes involved in the reorganization, the reorganization is then interrupted at this point.
To continue the reorganization, you must manually initiate the transition when the classes are no longer in use, by using the JADE development environment **Initiate Transition** command from the Schema menu **Reorg** command submenu or by running the **JadeReorgApp** application with the **initiateTransition** value specified for the **action** parameter.

**Notes** To ensure that no server applications are running, you can shut down the database server when the reorganization is interrupted prior to the transition. You can initiate the transition when the server is restarted. (For details, see "Initiating the Transition Phase of a Reorganization" under "Using the Reorg Command", or for details about doing so when running the **JadeReorgApp** application, see "Reorganizing the Database from the Command Line", in Chapter 3 of the **JADE Development Environment User’s Guide**.)

When performing a reorganization in production mode, the transition must be initiated in single user mode. For more details, see "Running JADE Production Mode Databases", in Chapter 1 of the **JADE Runtime Application Guide**.

**Replayable Reorganizations**

Replayable reorganizations enable a secondary system on a standby server to roll forward through a reorganization and have the effects of that reorganization applied.

When a reorganization is initiated on a primary database server, it must be possible for a secondary database to have the effects of the reorganization applied, which is made possible by auditing the reorganization operations in the database journal. This allows the reorganization to be **replayed** on the secondary database server when the journals are received.

The mechanism that is used to support replaying reorganizations on a secondary database can also be used to roll-forward through a reorganization when the database is not located on a primary server.

If archival recovery is enabled, you can use the database backup or recovery operation to roll-forward through a reorganization, as long as the appropriate journals are available.

You can replay reorganizations only when archival recovery is enabled (that is, the **EnableArchivalRecovery** parameter in the [PersistentDb] section of the JADE initialization file is set to **true**).

Replayable reorganizations must be restarted if a system failure results in a reorganization terminating. The reorganization is restarted at the last checkpoint established during the reorganization.

Restart an interrupted reorganization by using the **Restart Reorg** command from the Schema menu **Reorg** command submenu or the **JadeReorgApp** application and specifying the **restartReorg** value in the **action** parameter. (For details about restarting a failed reorganization, see "Restarting an Interrupted Reorganization" under "Using the Reorg Command", or for details about doing so when running the **JadeReorgApp** application, see "Reorganizing the Database from the Command Line", in Chapter 3 of the **JADE Development Environment User’s Guide**.)

For details about inhibiting the creation of .bak files during a reorganization, see "Location of Files In Reorganizations", later in this chapter.

If you are running in a Synchronized Database Environment (SDE), you should disable tracking on at least one secondary database while a reorganization is in progress on the primary database. The reorganization process on the primary database appends redo information to the journals, which continue to be transferred to attached secondary systems even when tracking is disabled. When the reorganization has been completed successfully on the primary system, you can then enable tracking, allowing the secondary system to replay the reorganization to regain synchronization with the primary database. If the reorganization fails for any reason, the secondary database remains synchronized with the primary database at the point prior to the reorganization being initiated.
Performing a non-replayable reorganization on a primary is recommended only for lengthy reorganizations in which re-cloning is the preferred option, as a non-replayable reorganization on a primary database cannot be replayed on a secondary database and you must therefore re-clone the secondary database from the primary.

Phases of a Reorganization

A reorganization of a JADE database incorporates the following phases.

1. Object Conversion
2. Transition
3. Relationship Maintenance
4. Commit Reorganization

For details, see the following subsections.

Object Conversion

Object conversion is the reorganization phase that coerces objects into a form that is compatible with the class definition in the latest schema version. This phase does not consider relationships between objects; rather structural changes affecting individual objects only are performed. While objects are being converted during this phase, applications can access and update objects defined by the current schema version without restriction.

The system remains available while the following changes are performed.

- Adding and removing attributes
- Adding and removing embedded references
- Adding and removing exclusive collections
- Changing the type of an attribute (any combination of type-to-type conversion)
- Changing the length of String or Binary attributes (includes converting embedded String or Binary primitive types to string large objects (slobs) or binary large objects (blobs), respectively, and the reverse, and truncation)
- Changing the precision or scale factor of a decimal attribute
- Moving blob and slob attributes up or down the class hierarchy
- Removing all inverses from a relationship (that is, after the change, there are no inverses between the properties that were involved in the relationship)

When all objects have been converted to their new state, any applications, including server applications, must be shut down that are accessing the schema or class being reorganized.

Fast Building of Collections

You can move the building or rebuilding of some ObjectSet and MemberKeyDictionary collections from the Relationship Maintenance phase to the Object Conversion phase of the reorganization process, by setting the FastBuildBTreeCollections parameter in the [JadeReorg] section of the JADE initialization file to true.

Fast building of collections significantly reduces the elapsed time of large database reorganizations involving collection maintenance, by using a faster extract, sort, and build algorithm and by allowing the collection maintenance to be performed in parallel by multiple reorganization workers using much less random I/O activity.
Chapter 14  Database Reorganization

Caution  Additional disk space up to three times the total size of the collections being built or rebuilt is required for extract and sort files.

Fast building of collections is available only for non-updating reorganizations. If you enable fast building of collections, you must disallow updates when you initiate the reorganization, as follows.

- In the JADE development environment, the Allow updates check box on the Classes Needing Reorg dialog is disabled and unselected.
- When loading a schema using the batch jadloadb utility, specify reorgAllowUpdates=false on the command line.
- When carrying out a reorganization using the non-GUI jadclient application with app=JadeReorgApp, specify reorgAllowUpdates=false on the command line.

Fast building or rebuilding of collections for a MemberKeyDictionary or ObjectSet reference occurs when the inverse is an existing object reference; that is, for a many-to-one relationship. The update mode of the collection reference must be automatic and that of the object reference must be manual. The ObjectSet reference includes subclasses of the collection class ObjectSet and subclasses of the collection class Set where the membership is a class. Member key dictionaries can have key paths and collection references with constraint methods.

Fast building of collections can be invoked for the following structural changes.

- An inverse is established between an existing MemberKeyDictionary or ObjectSet reference and an existing object reference.
- A MemberKeyDictionary or ObjectSet reference is added for an existing object reference inverse.
- The key definition of a MemberKeyDictionary reference is changed; for example, if:
  - Key elements are added, deleted, or reordered
  - The Sort Order, Case Insensitive, or Descending property of a key element is changed
- The constraint method of a MemberKeyDictionary reference is changed.
- A MemberKeyDictionary or ObjectSet class structure is changed; for example, the Entries Per Block property on the Tuning sheet.

Fast building of collections is not allowed when:

- The collection owner class is a subclass of JadeBytes.
- The collection membership is the JadeBytes class or a subclass.
- The object reference property is new and therefore has the initial default value of null.
  - If the inverse collection reference exists and is not empty, the offline collection builder populates the new reference property.
- The membership of the collection is not a class; for example, an ObjectSet with an interface as its membership.
- The collection reference has one or more inverses declared that are collections.

By default, the extract and sort files are placed in the database directory (that is, the directory containing the database control file). You can specify another location by using the ReorgSortDirectory parameter in the [JadeReorg] section of the JADE initialization file.
The dictionary key sort uses up to 50M bytes of memory for each reorganization worker, by default. You can increase the amount of sort memory to improve the sort performance when the total key length exceeds 30 bytes, using the `WorkerSortMemory` parameter in the `[JadeReorg]` section of the JADE initialization file.

Fast-built collections are densely packed, leaving two free entries in each block which gives an approximate load factor of 98 percent. For new collections, user-coded constructors are not executed and user-specified property values are initialized with `null` values. For existing collections that are rebuilt, user-specified property values are retained.

When a reorganization with the `FastBuildBTreeCollections` parameter set to `true` is replayed by a roll-forward recovery or on a secondary SDS or RPS node, the entire fast building process involving extracts, sorts, and builds does not occur. Instead, the collection headers and blocks audited during the original reorganization build jobs on the primary SDS node are applied from the journal by the reorganization worker after the objects have been converted and copied to the .reo files.

### Transition

The reorganization transition marks the point from which instances of versioned classes are no longer available to applications until the reorganization completes (or fails and is aborted). The transition can proceed only when no versioned schemas and classes are in use.

Updates to objects in user data files (versioned or unversioned) are not allowed after a transition has begun. Only the reorganization process is allowed to update files during the offline phase.

A reorganization can be configured to attempt to initiate the transition as soon as the object conversion phase is complete. Alternatively, the reorganization is suspended and the transition must be initiated by an explicit user action at a later stage.

**Note** If the transition cannot proceed because schemas and classes are in use, the reorganization log file lists the processes that are using these. In addition, you can use the Classes In Use Browser in the JADE development environment to identify the classes being used by each process.

### Relationship Maintenance

The relationship maintenance phase of a reorganization restores the referential integrity defined by the rules of the latest schema definition and includes the following actions.

- Adding an inverse
- Removing some, but not all, inverses from a relationship (that is, after the change, inverses still exist between the properties involved)
- Condition maintenance
- Inverse definition changes; for example, changing the inverse-required option
- Changes to dictionary specifications; for example, key definitions or the no-duplicates constraint
- Changes to the physical attributes of a collection; for example, block size and expected population
- Validation of the member objects of a collection against its member type definition
- Validation of reference property type changes
- Moving exclusive collections up or down the class hierarchy
**Note**  As these structural changes are not performed online, the database is not available to applications until the reorganization completes (or fails and is aborted).

## Commit Reorganization

During this final reorganization phase, the changes made by the reorganization are committed to the database and the schema version is updated. All changes then become available to other applications.

### Location of Files In Reorganizations

The following sets of files are involved in reorganizations and compactions.

- Original database files (.dat files)
- Reorganization work files (.reo files)
- Backups of the original database files (.bak files)

By default, the reorganization work files and the database backup files are located in the database directory. This requires significant free-space (at least double the size of the files being reorganized) to be maintained on the database volume.

Alternatively, you can use the following parameters in the JADE initialization file to manage the disk space requirements in a more-flexible way.

- **ReorgWorkDirectory** parameter in the [JadeReorg] section, which enables you to create reorganization work files on a volume other than the database volume.

- **ReorgBackupDirectory** parameter in the [JadeReorg] section, which enables you to locate the backup files on a volume other than the database volume.

  When a reorganization or compaction instantiates a file, it renames the database file (.dat) as a backup file (.bak), causing the file to move from the database directory to the directory specified in the ReorgBackupDirectory parameter and then renames the work file (.reo) to become the database file (.dat), causing that file to move from the directory specified in the ReorgWorkDirectory parameter to the database directory.

The optional **noReorgRecovery** parameter in the command line of the JADE non-GUI client (jadclient) or batch Schema Load utility (jadloadb) program specifies whether recovery is required if a reorganization fails. When you set this parameter to **true**, .dat files are removed rather than saved as .bak files. This minimizes disk space requirements at the cost of not being able to recover from a failure during the reorganization. The default value of this parameter is **false**.

**Note**  Use the noReorgRecovery parameter only when you deploy a system and are prepared to restore the system from backup if an error occurs during the deployment process.

## Reorganization Log Files

Details of the reorganization progress are output to a log file. Any errors that occur during the reorganization are logged to this file.
If a reorganization fails, you should examine the appropriate log file to identify the cause of any errors that must be addressed before restarting the reorganization. Schema instantiation can fail for various reasons; for example:

- The server could crash at any stage
- The inter-object phase could encounter a No duplicates violation
- A lock exception could be encountered in the commit phase

In all failure scenarios, committed transactions are not lost; that is, transaction durability is preserved. In other words, when you choose to perform part of an application deployment online and the deployment fails for any reason, user transactions that were successfully committed during the deployment can be recovered even if the effects of the failed deployment are removed.

In most cases, you can restart an interrupted schema instantiation, but this may entail redoing steps that have already been done.

Reorganization errors are output to the jomreorg.log file in the JADE log file directory. This directory is specified by the LogDirectory parameter in the [JadeLog] section of the JADE initialization file. For details, see "JADE Log Section [JadeLog]", in the JADE Initialization File Reference. In multiuser mode, the log file directory is defined by the settings on the server node.

### Correcting Data if a Reorganization Fails

Reorganization maintains a backup copy of all databases files involved in the reorganization. If the reorganization encounters an error, these backup files are used to restore the database.

When something goes wrong during a schema instantiation, you can perform one of the following actions.

- Restart the schema instantiation (by using the Restart Reorg command from the Schema menu Reorg command submenu or the JadeReorgApp application and specifying the restartReorg value in the action parameter, to restart the reorganization)
  
  This action applies to situations in which the schema instantiation was interrupted (for example, the machine halted) and reprocessing can still be accomplished in the remaining system downtime window.

- Abandon the schema instantiation (by using the Abort Reorg command from the Schema menu Reorg command submenu or the JadeReorgApp application and specifying the abortReorg value in the action parameter, to restore the database)
  
  This action may be necessary in failure situations when there is no degree of certainty the problem that caused the failure can be resolved and the instantiation completed within the remaining system downtime window.

If your reorganization fails because of a data error (for example, you have duplicate keys in a collection class that does not allow them or there is an invalid reference), the following error is most likely to be displayed.

```
3101 Unable to reorganize class - refer to jomreorg.log
```

Your reorganization log file contains the class in which the error occurred, to enable you to correct your data before actioning a successful reorganization, as follows.

1. Check your reorganization log file, to determine the class or classes that contains the error or errors.
2. Write some code to correct the data (for example, by using a method in the JadeScript class).
3. Initiate the reorganization of your schema again. (For details, see "Performing a Reorganization", in Chapter 3 of the JADE Development Environment User's Guide and the following section.)
Restarting Reorganizations

You must manually restart or abort all reorganizations that fail to complete (for example, because of a system crash).

You can always restart replayable reorganizations, which will continue from the last checkpoint established during the reorganization.

You can restart non-replayable reorganizations only if the system failure occurred while the reorganization was waiting for the transition to be initiated or if the failure occurred when the reorganization was committing. In all other cases, you must manually abort non-replayable reorganizations when the system is restarted, to restore the database to its pre-reorganization state.

When a database server node is restarted following a crash that occurred during a schema instantiation, restart recovery recovers all committed transactions including all transactions applied to the schema and forms definition file loads.

An interrupted database reorganization is not automatically restarted as part of database recovery. Restarting an interrupted reorganization requires an explicit action. (For details, see "Restarting an Interrupted Reorganization", in Chapter 3 of the JADE Development Environment User's Guide.)

Restarting an online reorganization interrupted during any of the online phases performs all file reorganizations again. Restarting an offline database reorganization reorganizes only files that have not been previously completed.

Aborting Reorganizations

To recover from a failed instantiation, you can restore from a prior database backup and then roll forward through available journals. See also "Aborting an Incomplete Reorganization", in Chapter 3 of the JADE Development Environment User's Guide.

Alternatively, you can abandon a failed instantiation by performing the following actions.

1. Abort the database reorganization by running a non-GUI jadclient application to execute JadeReorgApp from the command line.

   ```
   jadclient path=database-path
   ini=jade-initialization-file
   [server=multiUser|singleUser|readOnlyUser]
   schema=RootSchema
   app=JadeReorgApp
   startAppParameters
   action=abortReorg
   ```

   For more details, see "Performing a Reorganization" or "Reorganizing the Database from the Command Line", in Chapter 3 of the JADE Development Environment User's Guide.

2. Unversion schemas, by using the jadloadb batch Jade Schema Load utility or the JadeSchemaLoader application. (For details, see the JADE Schema Load User's Guide.)

   For simplicity, you can perform the first step unconditionally, even when there is no reorganization to abort.

   **Tip** Abandoning an instantiation using the above steps is generally preferable and in most cases it is quicker than restoring and recovering from a backup.
Reorganization Restrictions

The reorganization of your JADE database is restricted by:

- Features Restricted by the JADE Development Environment
- Unsupported Features
- Disk Space Requirements

These restrictions are described in the following sections.

Features Restricted by the JADE Development Environment

If instances of a class exist, JADE prevents the following actions from being performed.

- Changing the type of a collection; for example, from an array to a set.
- Changing the type of the keys of an ExtKeyDictionary class.
- Changing the membership of a collection to completely different types. You can change membership to a subclass but changing to a superclass is dependent on the existing instances.
- Moving instances from one database file to another; that is, changing the class map.

For details about using JCF commands to remap classes and move class instances, see "Using JCF Commands to Remap Classes and Move Class Instances", earlier in this chapter.

Unsupported Features

If instances of a class exist, the following actions can be performed, but there may be a loss of data when reorganization takes place.

- Recovery does not work across reorganizations.

  **Caution** When recovery is an issue, ensure that you perform a backup of the database before the reorganization is performed.

- Reorganization of primitive collections is not supported.

A warning message is displayed when a collection membership of primitive types is changed from one primitive type to another and the class is already marked for reorganization or the change will cause the class to be marked for reorganization, as the changing of collection membership of primitive types is not supported by the reorganization process.

Disk Space Requirements

When a reorganization is performed, the amount of disk space that is required on the drive that contains the database is not that of the complete database but a total of:

1. Twice the disk space of the data files (.dat files) that are being reorganized.
2. Additional disk space to allow for newly defined properties and properties with an increased size.
3. Additional disk space to allow for new inverses and rebuilt collections.
4. Additional disk space for journals, if the reorganization is replayable.

**SDS Reorganization Considerations**

This section contains the following topics.

- Structural Changes to Classes Without Instances
- Secondary Database Replay
- Secondary Database Availability

For details, see the following subsections.

**Structural Changes to Classes Without Instances**

When a structural change is made to a class on the primary that does not have instances, the class is not versioned; this applies to changes made in the JADE development environment and to schema loads performed using the Schema Load utility (that is, `jadload` and `jadloadb`) with the Only Structural Versioning load style.

**Note** If the class is part of an RPS mapping, it is versioned even if it has no instances. This is required for the RPS Datapump application to behave correctly.

Although the structural change does not cause a reorganization on the primary, a special audit record is written to the journal. When this audit record is replayed on the secondary, all user applications are shut down and sign-on is disabled in readiness for the transition step of the schema instantiation.

The reason for shutting down the applications is not because a reorganization is required, but because the applications must be restarted in order to use the new and changed class definitions.

To restart the applications on the secondary, a different audit record must be written to the journal on the primary. This is accomplished by running a non-GUI `jadclient` application on the primary, specifying the following:

```
jadclient path=database−directory ini=JADE−initialization−file−path server=multiUser schema=RootSchema app=JadeReorgApp startAppParameters action=auditEnableSecondaryApps
```

The replaying of this audit record on the secondary restarts all server applications.

For more details about automating the schema instantiation process, see "Reorganizing the Database from the Command Line", in Chapter 3 of the *JADE Development Environment User’s Guide*.

**Secondary Database Replay**

A secondary database automatically replays all phases of a schema instantiation, including any replayable database reorganization performed on the primary. A non-replayable reorganization, as the name implies, is not replayed by a secondary database; database tracking halts when it encounters the audit discontinuity record. (For details, see "Secondary Database Availability", in the following section.)
When database tracking has halted at an audit discontinuity record, query applications can still access the database although you will need to eventually re-establish the secondary database from a post-instantiation backup of the primary, to enable database tracking to continue beyond the discontinuity.

While a primary database is open and active with updating transactions in progress, journals continue to be generated and transferred to connected secondary servers. A secondary database enters a reorganization mode when it replays the first reorganization control record. The replay of a reorganization on a secondary processes through several states that can be interrogated by using the SDS Administration application. For details, see “Using the SDS Administration Application”, in Chapter 1 of the JADE Synchronized Database Service (SDS) Administration Guide.

The first reorganization replay state is Seeking Approval. While in the Seeking Approval state, database tracking scans ahead through journals looking for specific reorganization control records, without replaying any reorganization operations or database updates. Essentially, replay is paused until the disposition of the reorganization process on the primary has been determined. Tracking remains in the Seeking Approval state until an initiate transition or an abort reorganization audit record is processed.

The reorganization proceeds only when an initiate transition audit record is processed. This record signifies that the online phases completed on the primary and an initiate transition audit record was executed. However, if an abort reorganization audit record is encountered, the secondary splices out the failed reorganization and continues with normal replay, applying any updates that occurred during the failed or aborted reorganization.

A structural change to a class made in the JADE development environment on the primary system or loaded using the Schema Load utility (that is, Jadload and Jadloadb) with the Only Structural Versioning load style does not cause it to be versioned if it has no persistent instances. However, a special audit record is output to the journal.

**Secondary Database Availability**

Replay of a schema instantiation on a secondary database has an online and an offline phase. When an online reorganization is replayed on an SDS secondary, the database remains available to inquiry applications for the duration of the online phase of the schema instantiation. Inquiry applications can continue to access objects defined by the current schema version.

When the initiate transition audit record is replayed, the database tracker performs the offline phase of the instantiation, accomplishing a transition to the latest version of the schema definition on the primary database. Initiating a transition has the same prerequisites as it did on the primary; that is, no versioned schemas and classes can be in use. To achieve this, JADE attempts to gracefully shut down all user applications, including server applications, before proceeding.

As the instantiation does not proceed until all user applications have been shut down, if a user application has stalled and is unable to terminate, the instantiation cannot proceed. To resolve a stalled instantiation, an administrative user must intervene to arrange for applications to be terminated and nodes to be shut down, if required. If necessary, you can shut down and restart the database server at this point.

When the schema instantiation has been replayed and committed, database server and application server applications configured to start automatically are restarted.
The following image shows the phases occurring on the secondary database.

By default, the initiate transition audit record is replayed as soon as it is encountered. If you need to schedule when this occurs so that you can arrange for a graceful termination of user applications, consider initiating a managed transition. The main difference in a managed transition is that a stop tracking audit record is inserted in the audit stream. This causes secondary database tracking to halt just prior to starting the offline phase so that the required downtime for inquiry applications can be managed.

The following image shows the phases in a managed transition.

Deferring replay by disabling tracking allows a failed schema instantiation to be spliced out on both the primary and secondary without the need to re-establish the secondary from a backup.

For details about starting applications in a secondary system and enabling user sign-on, see “Stopping and Starting Server Applications in Secondary Systems”, in Chapter 10.
Chapter 15  Using the Messaging Framework

This chapter contains the following topics.

- Overview
- Components of the Messaging Framework
- Messaging Terminology
- Concept of Queued Messaging
- Classes in the Messaging Framework
- Examples of the Use of the Messaging Framework
- Message Payload
  - Unstructured Data
  - Structured Data
- Specific Transports
  - JADE Messaging Transport
  - WebSphere Messaging Transport
- Generic Message Example

Overview

It is unusual to find an application that does not need to exchange data with other applications. In addition, it is becoming common for this data exchange to happen automatically across digital communication links (for example, the Internet). A large number of these interfaces still use custom designs, so that new users of an interface frequently need to write all or part of the exchange mechanism from scratch rather than using a standard pre-written module. SOAP over HTTP is a well-defined open standard for data exchange, which forms the basis for Web services. For more details, see the Web Services white paper on the JADE Web site at https://www.jadeworld.com/developer-center/resource-library/white-papers.

IBM and Microsoft offer proprietary message queuing subsystems (WebSphere MQ and Microsoft MQ, respectively), which are available for multiple operating systems and hardware architectures. However, these queuing subsystems require considerable experience to implement.

The messaging framework is a basic generic message exchange module that provides a wrapper for the standard IBM WebSphere MQ message queuing transport and for a JADE message queuing transport.

The generic module interface provides a high performance subsystem that enables developers to:

- Handle a wide range of underlying transports within a common framework.
- Write code that works regardless of the transport chosen at run time. (Some transports require a specific code structure to work correctly; notably transaction integration.)
- Switch between transports without changing any logic other than the queue name.
- Access transport-specific features.
Access standard external transports (for example, IBM WebSphere MQ).
Minimize the amount of JADE code written.
Use an event-driven processing model.

Components of the Messaging Framework

The main components of the messaging framework are:

- **Messaging**
  Transfer of a user-supplied package of data (the payload) from one process to another. The sender and receiver processes must be executing concurrently and have a channel established between them (for example, a TCP/IP connection).

- **Message queuing**
  Messages are delivered to a location from which the receiver retrieves them when required. This usually requires a third process (the queue manager).

Messaging Terminology

The following table explains terminology that is commonly used in messaging systems.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute queue name (or full queue name)</td>
<td>A string that fully identifies a particular queue. It includes transport type, host address, queue manager name, and queue name.</td>
</tr>
<tr>
<td>message</td>
<td>A data packet that includes a set of properties (the header) and a payload.</td>
</tr>
<tr>
<td>payload</td>
<td>The user supplied data block. Transports can have restrictions on the length of the payload.</td>
</tr>
<tr>
<td>queue</td>
<td>The entity to which messages are sent and from which messages are retrieved.</td>
</tr>
<tr>
<td>queue manager</td>
<td>A non-application process that is responsible for receiving messages from senders (local and remote) and storing them until they are retrieved by an application.</td>
</tr>
<tr>
<td>receive or get</td>
<td>To remove a message from a queue.</td>
</tr>
<tr>
<td>send or put</td>
<td>To insert a message into a queue.</td>
</tr>
<tr>
<td>system</td>
<td>A database server and its associated client nodes.</td>
</tr>
<tr>
<td>transport</td>
<td>A messaging implementation.</td>
</tr>
</tbody>
</table>

The queue manager can have the following additional responsibilities.

- Communicating with other queue managers
- Persistent storage of queue definitions
- Persistent storage of unprocessed messages
- Generating and sending reports about messages that have been discarded, delivered, and retrieved
- Initiating an application when a message is placed into a queue (for example, WebSphere MQ triggers)

**Note** WebSphere MQ allows multiple queue managers on a host.
Concept of Queued Messaging

The following diagram shows a process inserting a message into a queue and a second process retrieving a message.

![Diagram of a process inserting a message into a queue and a second process retrieving a message.]

A queue is an ordered list of messages. The messages are handled by first-in-first-out algorithm but some transports provide ways of retrieving messages out of order. Additionally, some transports support message priorities.

The queue decouples the sender and receiver, enabling messages to be sent when the receiver is not active. It enables messages to be received and processed at a later time.

There can be a number of receivers active or no receivers active. For example, a queue can be allowed to build up during the day, and its messages can be processed during the night.

It is usual for a process to create a unique queue for replies. If the queue is created using the `openQueue` method of the `JadeMessagingFactory` class with a masked queue name, the resulting queue is guaranteed to have a unique name.

A sender has no direct control over which (if any) receiver retrieves and processes a specific message.

Messages are sent to a queue manager. The receiver retrieves messages from the queue manager.

Classes in the Messaging Framework

The classes that comprise the JADE messaging framework are listed in the following table.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JadeGenericMessage</td>
<td>Encapsulates the building and analysis of messages</td>
</tr>
<tr>
<td>JadeGenericMessagingIF</td>
<td>Defines message arrival and queue management callback methods</td>
</tr>
<tr>
<td>JadeGenericQueue</td>
<td>Encapsulates a destination for the transmission and retrieval of messages</td>
</tr>
<tr>
<td>JadeGenericQueueManager</td>
<td>Encapsulates the interface to a queue manager</td>
</tr>
<tr>
<td>JadeMessagingException</td>
<td>Defines the behavior of exceptions that arise when using the messaging framework</td>
</tr>
<tr>
<td>JadeMessagingFactory</td>
<td>Encapsulates the behavior for creating and opening messaging queues</td>
</tr>
</tbody>
</table>

For details, see Volume 1 of the JADE Encyclopaedia of Classes.

Examples of the Use of the Messaging Framework

This section contains a number of simplified examples that illustrate the use of the messaging framework to accomplish common messaging tasks using generic features. In this example, a datagram message is being sent to an existing queue and a reply is not expected.
To send a datagram message

1. Create an instance of the `JadeMessagingFactory` class. This class encapsulates the behavior for creating and opening queues.

2. Create and open a queue by calling the `openQueue` method on the `JadeMessagingFactory` instance created in the previous step. The method returns an instance of the `JadeGenericQueue` class configured for the specified transport.

3. Create an empty message by calling the `createMessage` method on the `JadeGenericQueue` instance created in the previous step. The method returns an instance of the `JadeGenericMessage` class configured for the specified transport.

4. Optionally set properties for the message object.

5. Optionally assign the payload (that is, the content) of the message. The other way of sending information is by setting user properties.

6. Insert the message in the queue by calling the `putMessage` method on the `JadeGenericQueue` instance.

The following example shows a datagram message being sent using the `JadeMQ` message transport.

```java
vars
    factory : JadeMessagingFactory;
    queue : JadeGenericQueue;
    message : JadeGenericMessage;
begin
    // Step 1 - create the factory object
    create factory transient;
    // Step 2 - create and open the queue
    queue := factory.openQueue("JadeMQ://localnode/MainQ", "MustExist");
    // Step 3 - create the message
    message := queue.createMessage(true);
    // Step 4 - optionally set message properties
    message.type := JadeGenericMessage.Type_Datagram;
    // Step 5 - set message content
    message.body := "Hello World".Binary;
    // Step 6 - insert message in queue
    queue.putMessage(message, "");
epilog
    delete message;
    delete queue; // implicitly closes the queue
    delete factory;
end;
```

To retrieve a message from an existing queue

1. Create an instance of the `JadeMessagingFactory` class. This class encapsulates the behavior for creating and opening queues.

2. Create and open the queue containing the message, by calling the `openQueue` method on the `JadeMessagingFactory` instance created in the previous step. The method returns an instance of the `JadeGenericQueue` class.

3. Create an empty message, by calling the `createMessage` method on the `JadeGenericQueue` instance created in the previous step. The method returns an instance of the `JadeGenericMessage` class.

4. Retrieve the message in the queue, by calling the `getMessage` method on the `JadeGenericQueue` instance. The payload of the message can be cast to the type of data it contains.
The following example shows a message being retrieved from a **JadeMQ** message queue.

```java
vars
    factory : JadeMessagingFactory;
    queue : JadeGenericQueue;
    message : JadeGenericMessage;
begin
    // Step 1 - create the factory object
    create factory transient;
    // Step 2 - create and open the queue
    queue := factory.openQueue("JadeMQ://localnode/MainQ", "MustExist");
    // Step 3 - create a message object
    message := queue.createMessage(false);
    // Step 4 - retrieve a message
    queue.getMessage(message, "");
    write message.body.String;
epilog
    delete message;
    delete queue; // implicitly closes the queue
    delete factory;
end;
```

### To reply to a message

In the example in this section, a message is received that has the **replyQueueFullName** property set. This property is set to instruct the receiver of the message to send any reply to the specified queue. To assist the sender of the original message in matching the original message with the reply message, the **correlationID** property of the reply message is set to the **messageID** property of the original message.

1. Create and open the reply queue, by calling the **openQueue** method.
2. Create an empty message, by calling the **createMessage** method on the **JadeGenericQueue** instance created in the previous step. The method returns an instance of the **JadeGenericMessage** class.
3. Set properties for the reply message object.
4. Assign the payload (that is, the content) of the reply.
5. Insert the reply in the queue, by calling the **putMessage** method on the **JadeGenericQueue** instance.

The following code fragment shows a reply message being constructed and sent.

```java
// Step 1 - create and open the reply queue
replyQueue := factory.openQueue(message.replyQueueFullName,"MustExist");
// Step 2 - create the reply message object
reply := replyQueue.createMessage(true);
// Step 3 - set correlationID and message type properties
reply.correlationID := message.messageID;
reply.type := JadeGenericMessage.Type_Reply;
// Step 4 - set reply message payload
reply appendString("Acknowledging receipt");
// Step 5 - insert reply message in reply queue
replyQueue.putMessage(reply, "");
```

### Message Payload

The most important part of the message is its payload (that is, the content of the message).
The payload can be unstructured binary or text, or it can be structured in the form of name-value tuples.

**Note** You cannot mix structured and unstructured data within a single message.

### Unstructured Data

The payload can be set by direct assignment to the `body` property the message. Alternatively, the payload can be set indirectly by using the methods of the `JadeGenericMessage` class listed in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>appendBinary</code></td>
<td>Adds the specified binary information to the end of the message body</td>
</tr>
<tr>
<td><code>appendString</code></td>
<td>Adds the specified text to the end of the message body</td>
</tr>
<tr>
<td><code>appendStringAsUtf8</code></td>
<td>Encodes the specified text as UTF8 and adds to the end of the message body</td>
</tr>
<tr>
<td><code>appendUtf8</code></td>
<td>Adds the specified UTF8 text to the end of the message body</td>
</tr>
</tbody>
</table>

Before content is added, the message object must be initialized for sending. Do this by calling the `createMessage` method of the `JadeGenericQueue` class with the `forPUT` parameter set to `true`, or by calling the `initializeForPut` method of the `JadeGenericMessage` class on an existing message object.

The following code fragment shows a message being constructed from a text string.

```java
// initialize the message
message := queue.createMessage(true);
// describe the type of data in the payload
message.format := "text/UTF8";
// build message payload
message.appendStringAsUtf8("Hello World");
message.appendStringAsUtf8("Glad to be here");
// insert message in queue
queue.putMessage(message, ";");
```

When the message is retrieved, the `format` property can be examined to determine the type of data in the payload. The `body` property, which is of type `Binary`, can then be cast to the correct type. In this case, it would be more efficient to use the `getUtf8BodyAsString` method of the `JadeGenericMessage` class.

```java
if message.format = "text/UTF8" then
    // str := message.body.String;
    str := message.getUtf8BodyAsString;
end;
```

### Structured Data

The payload is set by using the `appendBodyTuple` method of the `JadeGenericMessage` class.

Before content is added, the message object must be initialized for sending. Do this by calling the `createMessage` method of the `JadeGenericQueue` class with the `forPUT` parameter set to `true`, or by calling the `initializeForPut` method of the `JadeGenericMessage` class on an existing message object.

The `body` property contains a series of named values (tuples). The name of a tuple is case-sensitive and must start with a letter. The subsequent characters (up to a maximum of 89 characters) can be letters, digits, underscores, or periods; for example, valid tuple names could be `cust.name`, `cust.address`, and so on.
One of the main uses of tuples is to make it easy to serialize an object by copying its properties into a message. Most primitive type property values are converted to UTF8 text apart from Binary values, which do not need to be converted and are sent as is. References are handled by serializing the oid value.

The following code fragment shows a message being constructed from name-value tuples.

```java
// initialize the message
message := queue.createMessage(true);
// describe the type of data in the payload
message.format := "JadeTpls";
// set message payload
message.appendBodyTuple("color", "Green");
message.appendBodyTuple("height", 200);
message.appendBodyTuple("width", 300);
message.appendBodyTuple("new", false);
// insert message in queue
message.inspectModal;
queue.putMessage(message, null);
queue.putMessage(message, null);
```

The first time the `appendBodyTuple` method is called, the `format` property is automatically set to `JadeTpls`.

When the message is retrieved and the value of the `format` property is set to `JadeTpls`, the `getBodyTuple` method of the `JadeGenericMessage` class can be used to scan the text in the message body for a tuple with the specified name, as shown in the following code fragment. The method returns `false` if there is no tuple with the specified name.

```java
reply.getBodyTuple("color", any);
// value of 'color' tuple is any.String
reply.getBodyTuple("height", any);
// value of 'height' tuple any.Integer
reply.getBodyTuple("new", any);
// value of 'new' tuple any.Boolean
```

### Specific Transports

The messaging framework provides much functionality that is common to the supported transports: the JADE messaging transport and the WebSphere MQ messaging transport.

The following subsections cover functionality that is specific to each supported transport.
JADE Messaging Transport

The following diagram shows two processes accessing a JadeMQ queue.

The processes must be running from a single JADE system; that is, the queuing mechanism is not intended as a way for one JADE system to communicate with another. Additionally, both processes must be running within the same node. The applications corresponding to these processes could be running from different schemas.

The queue and queue manager are implemented within the address space of a single JADE node. Only JADE processes within that node can access the queue.

The queue can be created by the process that sends the message or receives the message. A sender can put messages into a queue that has no receiver (that is, a queue that has not been opened for get operations).

The message list for a queue is stored in memory only. Message handling is unreliable in that the queue (and any messages in the queue) disappears when it is deleted, when the queue manager is deleted, or when the node terminates. If a receiver aborts while processing a message, the message is not re-queued.

When a queue has been created, it remains available until it is deleted by the last user closing it. A user process must recreate the queue each time a node is restarted.

The JADE messaging transport is intended for the following purposes.

- Passing messages between processes in the same node.
- Passing messages within a single process. For example, GUI changes are queued until after a transaction has completed.

The following extensions are specific to the JadeMQ transport.

- Messages can have user-defined message properties.
- When retrieving messages by using the getMessageByCorrelationID method of the JadeGenericQueue class, up to 64 identifiers can be specified.
- A timeout option can be specified when sending a message.

Restricting Access to Queues

If you create a queue with Access=Private specified as part of the options parameter, only the creating process can reopen it. You can specify the AccessPassword open option to limit access to a queue to specific parts of an application (for example, limiting access to within a package).

If you create a queue with Access=Protected specified as part of the options parameter, all subsequent attempts to open the queue must include an AccessPassword option value that matches exactly the value specified when the queue was created. If the AccessPassword option was not specified, the queue behaves as if Access=Public had been specified, and can be opened by any process that knows the name of the queue.

You can use the generateAccessPassword method of the JadeMessagingFactory class to generate unique, random, non-repeating 32-character passwords.
WebSphere Messaging Transport

To use the WebSphere messaging transport:

- A copy of the WebSphere MQ client support software must be obtained and licensed from IBM or an IBM reseller
- The RegisterIBMWMQ parameter in the [JadeGenericMessaging] section of the JADE initialization file must be set to true

A WebSphere administrator defines queue managers and queues. An application can only use queues that have been defined previously. However, you can create a temporary queue (for example, for reply messages) that is based on a previously declared model queue with masking.

The following diagram, which is only one possible scenario, shows JADE processes in different nodes accessing a WebSphere MQ queue.

Each JADE process, which could be on the same node or a different node, is using the WebSphere client interface to communicate with the WebSphere messaging hub. Either of the JADE processes could be replaced by some other form of WebSphere client. In general, the architecture supports communication across JADE nodes and hosts.

The queue manager is a separate, active, operating system process. Unconsumed messages can be lost when the queue manager terminates, depending on the configuration. More than one queue manager is allowed in a machine. A queue manager is a global resource, and any process or thread on that machine can open a queue owned by that queue manager (allowing for security restrictions).

Messages are transferred between queue managers using point-to-point connection-based protocols. The queue managers can be on the same or different machines, which can be geographically distributed.

When a sender passes a message to a local queue manager for forwarding to a queue owned by another queue manager, the sender does not need to manage the underlying transport (for example, by opening a TCP/IP connection). Each queue manager can be addressed separately, and therefore each queue can be addressed separately. A queue hosted by another queue manager is declared locally as a remote queue and is linked to the transmission queue for a channel connection between the two queue managers.

Queues and queue managers are defined persistently, but the message list for a queue is usually stored in memory only. The JADE implementation uses unreliable message handling. Messages waiting in the queue are lost when the queue manager terminates. A get operation permanently removes a message from the queue. A put operation releases the message for immediate delivery.

A receiver gets an exception while waiting for messages if the queue manager terminates. A sender gets an exception if it attempts to put a message in a queue owned by a terminated queue manager to which it was directly linked. A sender can successfully put a message in a remote queue when the owner queue manager is not active; the local queue manager holds the message in memory until it can be forwarded.

The following extensions are specific to the WebSphere MQ transport.

- Reports can be requested from the receiving queue manager providing information about the confirmation of arrival and delivery of messages. For example, the queue manager can send a report message to the
sending application when it *puts* the message in the target queue or when the application *gets* it from the queue.

- Inter-node and inter-host messaging enables messages to be passed between processes that may be in separate JADE nodes or on separate machines. The processes may even belong to separate JADE systems.
- Communication with non-JADE applications is possible.

### Opening a WebSphere MQ Queue

When you open a WebSphere MQ queue you must include an MQSERVER clause as part of the `openQueue` method options, either embedded in the `fullName` parameter or as one of the options. This provides information for the WebSphere MQ client interface to connect to the specified queue manager. For details about the client interface, see the *WebSphere MQ Client Manual*.

The format of the `fullName` parameter, including an MQSERVER clause, is as follows.

```
IBMWMQC://QMNAME/QUEUE1;?MQSERVER='CHNNAME/TCP/HOSTFQDN(PORTNUM)'
```

The parts of the `fullName` parameter are listed in the following table.

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QMNAME</td>
<td>The name of the queue manager.</td>
</tr>
<tr>
<td>QUEUE1</td>
<td>The name of the queue.</td>
</tr>
<tr>
<td>CHNNAME</td>
<td>The name of the MQI (or server connection) channel.</td>
</tr>
<tr>
<td>HOSTFQDN</td>
<td>The fully qualified domain name of the host where the queue manager resides.</td>
</tr>
<tr>
<td>PORTNUM</td>
<td>The TCP port number associated with the channel.</td>
</tr>
</tbody>
</table>

The following example shows a `fullName` parameter that specifies the opening of an existing queue.

```
IBMWMQC://QM1/QUEUE1;?MQSERVER='CHNL1/TCP/george(32001)'
```

The following example shows a `fullName` parameter that specifies the opening of a temporary reply queue. It uses a *model queue* definition.

```
IBMWMQC://QM1/REP*;?MQSERVER='CHNL1/TCP/george(32001)';Model=MODELREPLYQ
```

### Name Lengths

WebSphere MQ limits queue and queue manager names to 40 characters.

A masked queue name is limited to 100 characters before the asterisk.

### Message Format Property

WebSphere MQ limits the `format` property to eight characters.

Names beginning *MQ* in uppercase, lowercase, and mixed case have meanings that are defined by the queue manager; do not use names beginning with these letters for your own formats.

For details about the queue manager built-in formats, see the *WebSphere MQ Application Programming Reference* document.
Message messageID Property

WebSphere MQ generates values containing all possible byte values including embedded nulls (bytes with a value of zero) for the messageID property.

WebSphere-Specific Errors

The WebSphere MQ application programming interface (API) returns two values describing the success or failure of each call. The first value is the completion code and can be one of the following values:

- MQCC_OK (0), indicating that the call completed successfully.
- MQCC_WARNING (1), warning that the call was only partially completed.
- MQCC_FAILED (2), indicating that the call failed.

The second value is the reason code. If the value of the completion code is MQCC_OK, the value of the reason code is MQRC_NONE (0). The other completion code values have many different reason code values, which are described in the WebSphere MQ Messages document.

The JADE messaging framework maps some common reason codes to specific JADE error codes. For example, the openQueue error MQRC_Q_MGR_NAME_ERROR (2058) is mapped to error 31789 (QueueManager is unknown) and MQRC_UNKNOWN_OBJECT_NAME (2085) is mapped to error 31738 (Queue does not exist).

The remaining errors are mapped to JADE error code 31785 (IBMWMQ exception) and the completion code and reason code are included as part of the errorItem text for the exception. For example, an unknown channel name specified in the openQueue option MQSERVER clause causes exception 31785 with an errorItem property of “completion=2, reasonCode=2059”.

Generic Message Example

In the examples in the following subsections, there are separate processes running on the messaging server and messaging client machines. The process running on the server receives requests and generates replies. The process running on the client sends a request and waits for a reply.

The application is event-driven and has a callback method that wakes up the application when a request message is inserted into the request queue.

The code examples are taken from a sample messaging system, the schema files for which are located in the messagingFramework subdirectory of the examples directory on the JADE release medium.

Server Application

The following properties are defined on the main form for the server application.

- myMessagingFactory of type JadeMessagingFactory
- myRequestQ of type JadeGenericQueue
- myRequestMsg of type JadeGenericMessage

When the application initializes, the default request queue is automatically opened.

To prepare to receive messages

1. Create an instance of the JadeMessagingFactory, if one does not exist.
2. Open the request queue.
3. Enable event callbacks.
4. Set the value of the `defaultGetMessageOptions` property for the queue.

The following code fragment implements this algorithm.

```java
if self.myMessagingFactory = null then
    self.logIt("Creating MessagingFactory...");
    create self.myMessagingFactory transient;
endif;
// If the queue does not already exist then create it as a public queue
openopts := "Access=Public;Create=Missing;Usage=Get,Inq";
self.logIt("Opening queue " & self.queueName & ", options=" & openopts & "...");
self.myRequestQ := self.myMessagingFactory.openQueue(self.queueName,/openopts);
if self.myRequestQ = null then
    // Probably got an UnhandledException before we got here.
    self.logIt("**Queue open failed");
else
    self.logIt("opened successfully = " & self.myRequestQ.fullName);
    // Ask for event callbacks when messages arrive in our queue.
    self.myRequestQ.notifyEventsAsync(self, JadeGenericQueue.Notify_Continuous);
    // Set default Get options to NoWait because we are using event-driven
    // processing and do not want to hang if another process has got all the
    // messages. This is more efficient than specifying the same options on
    // each getMessage call.
    self.myRequestQ.defaultGetMessageOptions := "NoWait";
    self.logIt("configured and accepting events");
    self.caption := "Server - " & self.myRequestQ.name;
endif;
```

To process a request and send a reply

**Note** The message processing occurs in the implementation of the `JadeGenericMessagingIF` interface `messageArrivedEvent` method. In this example, a reply containing the text of the body of the request converted to uppercase is sent.

1. Create or initialize the request message.
2. Get the first message from the queue. The value of the `defaultGetMessageOptions` property is set to `NoWait`. If the queue is empty (because another process has already removed the message), the `getMessage` method immediately returns `false`.
3. Check that the value of the `type` property of the message is `Type_Request`.
4. Open the reply queue specified by the value of the `replyQueueFullName` property of the message.
5. Create and initialize the reply message.
6. Process the request and update the reply. (Convert the body of the request message to uppercase and copy it into the body of the reply message.)
7. Put the reply message into the reply queue.
8. Delete the reply message and reply queue, which implicitly closes the queue.
The following example implements this algorithm.

```java
vars
gotMsg : Boolean;
rqstMsg : JadeGenericMessage;
replyQ : JadeGenericQueue;
replyMsg : JadeGenericMessage;
msgbody : String;
openopts : String;
begin
    self.logIt("MessageArrived: queue=" & queue.fullName);
    // Create or re-initialize our request message for get.
    if self.myRequestMsg = null then
        self.myRequestMsg := queue.createMessage(false);
    else
        self.myRequestMsg.initializeForGet();
    endif;
    // local variable is cheaper than dereferencing self.
    rqstMsg := self.myRequestMsg;
    gotMsg := queue.getMessage(rqstMsg, "");
    if not gotMsg then
        // Another process beat us to the message
        self.logIt(" no message to get");
        return;
    endif;
    self.logIt(" msgId = " & rqstMsg.messageID.toGraphicString(99));
    self.logIt(" received from " & rqstMsg.senderID);
    // Ensure the sender set the message type
    if rqstMsg.type <> JadeGenericMessage.Type_Request then
        self.logIt(" unexpected msg.type = " & rqstMsg.type.String);
        return;
    endif;
    self.logIt(" replying to " & rqstMsg.replyQueueFullName);
    openopts := "Create=Never; Usage=Put";
    replyQ := self.myMessagingFactory.openQueue(rqstMsg.replyQueueFullName, openopts);
    if replyQ = null then
        self.logIt(" reply queue open failed = " & rqstMsg.replyQueueFullName);
        return;
    endif;
    replyMsg := replyQ.createMessage(true);
    replyMsg.type := JadeGenericMessage.Type_Reply;
    replyMsg.correlationID := rqstMsg.messageID;
    // Uppercase request body for reply
    msgbody := rqstMsg.getUtf8bodyAsString().toUpper();
    replyMsg.appendStringAsUtf8(msgbody);
    replyQ.putMessage(replyMsg, "");
    self.logIt(" reply sent = " & replyMsg.messageID.toGraphicString(99));
epilog
    delete replyQ;
    delete replyMsg;
end;
```
The server application also offers the following functionality, which is not examined in detail.

- Change the name of the queue
- Close the queue
- Open the queue
- Disable and enable event callbacks

### Client Application

The following properties are defined on the main form for the client application.

- `myMessagingFactory` of type `JadeMessagingFactory`
- `myRequestQ` of type `JadeGenericQueue`
- `myRequestMsg` of type `JadeGenericMessage`
- `myReplyQ` of type `JadeGenericQueue`
- `myReplyMsg` of type `JadeGenericMessage`

#### To send a request message

1. Open the reply queue if it is not already open.
2. Create or initialize the reply message.
3. Open the request queue if it is not already open.
4. Create or initialize the request message.
5. Construct the request message.
6. Put the request message into the request queue.

The following code fragment implements this algorithm.

```java
if self.myReplyQ = null then
    self.mnuFileRplyOpen_click(self.mnuFileRplyOpen);
    if self.myReplyQ = null then
        return;
    endif;
endif;
if self.myReplyMsg = null then
    self.myReplyMsg := self.myReplyQ.createMessage(false);
else
    self.myReplyMsg.initializeForGet();
endif;
if self.myRequestQ = null then
    self.mnuFileRqstOpen_click(self.mnuFileRqstOpen);
    if self.myRequestQ = null then
        return;
    endif;
endif;
if self.myRequestMsg = null then
    self.myRequestMsg := self.myRequestQ.createMessage(true);
else
```
To open the reply and request queues

The following steps are required to open a reply queue. (By changing the names within the code, a reply queue could also be opened.)

1. Obtain the queue name from a text box.
2. Creates an instance of the JadeMessagingFactory, if one does not exist.
3. Open the queue.

The following example implements this algorithm.

```python
vars
qname : String;
begin
qname := self.txbReplyQueue.text.trimBlanks();
if qname = "" then
    self.logIt("***Reply Queue name is blank");
    return;
endif;
if self.myMessagingFactory = null then
    self.logIt("Creating MessagingFactory...");
    create self.myMessagingFactory transient;
endif;
self.logIt("Opening REPLY queue " & qname & "...");
// If the queue does not already exist then create it as a public queue
self.myReplyQ := self.myMessagingFactory.openQueue(qname,
    "Access=Public;Create=Missing");
if self.myReplyQ = null then
    // Probably got an UnhandledException before we got here.
    self.logIt("**Open failed");
else
    self.logIt(" opened successfully = " & self.myReplyQ.fullName);
end;
```

To receive a reply from the queue

1. Open the reply queue, if it is not already open.
2. Create or initialize the reply message.
3. Get the first reply message. A ten-second timeout is specified so that the client does not wait indefinitely for
the server to send a reply.

When the reply is eventually sent, it stays in the reply queue until the `getMessage` method is invoked again.

The following example implements this algorithm.

```java
vars
    gotOne : Boolean;
begin
    if self.myReplyQ = null then
        self.mnuFileRplyOpen_click(self.mnuFileRplyOpen);
        if self.myReplyQ = null then
            return;
        endif;
    endif;
    if self.myReplyMsg = null then
        self.myReplyMsg := self.myReplyQ.createMessage(false);
    else
        self.myReplyMsg.initializeForGet();
    endif;
    // Get reply but do not wait forever
    gotOne := self.myReplyQ.getMessage(self.myReplyMsg, "Timeout=10000");
    self.logIt("Reply:");
    if not gotOne then
        self.logIt(" getMessage() returned false");
        return;
    endif;
    self.logIt("  MsgId = " &
                self.myReplyMsg.messageID.toGraphicString(99));
    self.logIt("  CorrelId = " &
                self.myReplyMsg.correlationID.toGraphicString(99));
    self.logIt("  Body = " & self.myReplyMsg.body.toGraphicString(120));
end;
```
Chapter 16 Using Asynchronous Method Calls

This chapter contains the following topics.

- Overview
- Asynchronous Worker Applications
- Making an Asynchronous Method Call
  - Preparing for an Asynchronous Method Call
  - Invoking the Asynchronous Method Call
  - Waiting for Asynchronous Method Calls to Complete
  - Obtaining Results from an Asynchronous Method Call
- Example of Using Asynchronous Method Calls in a Text Search

Overview

Asynchronous method calls enable an application to invoke methods in parallel, to reduce the elapsed time taken to process the business transaction. Asynchronous method calls use the JADE messaging framework. For details, see Chapter 15, "Using the Messaging Framework".

The following steps are required in your logic to perform an asynchronous method call.

1. Create an instance of the JadeMethodContext class.
2. Set the object properties in your JadeMethodContext instance (for example, the workerAppName and timeout properties).
3. Initiate the asynchronous method call, by using the invoke method of the JadeMethodContext class.

   Note At least one worker application must be running with a name matching the value of the workerAppName property.

4. Continue processing until the results of the asynchronous method call must be harvested.
5. Wait for the call to complete, by using the waitForMethods method of the Process class.
6. Check for call errors (that is, exceptions).
7. Examine the call result value.

The Process class provides the waitForMethods method, whose methodContextList parameter value consists of one or more JadeMethodContext instances or an instance of the ObjectArray class that contains only JadeMethodContext instances. The waitForMethods method suspends the process until one of the specified method contexts completes or times out.
Asynchronous Worker Applications

An asynchronous worker application is one that executes a method call that originates from another application running in the same node. The worker application must be defined in the schema in which the asynchronous method call is made. The information about the receiver, method name, and parameters is encapsulated in a JadeMethodContext instance.

The worker application must execute the asyncInitialize method on the Application class so that the internal structures required to process asynchronous method calls can be constructed and the application can be prepared to receive and process requests. This method is typically called from the initialize method for a worker application that processes asynchronous method calls.

The worker application can also execute the asyncFinalize method of the Application class to delete internal transient objects used in processing asynchronous method calls. The internal data structures are released after a worker terminates, even when this method is not called. This method is typically called from the finalize method for a worker application that processes asynchronous method calls.

The value of the applicationType property of the worker application can be ApplicationType_GUI or ApplicationType_Non_GUI. Requests are directed to a worker application by setting the workerAppName property of the JadeMethodContext instance to the name of the worker application. A number of copies of the worker application with the same name share a request queue and an asynchronous method call request is executed by the first worker application that becomes available.

The following examples show the coding of initialize and finalize methods for an asynchronous worker application.

```plaintext
awInitialize();
begin
  app.asyncInitialize;
end;

awFinalize();
begin
  app.asyncFinalize;
end;
```

Making an Asynchronous Method Call

The JadeMethodContext instance stores the information and provides the interface to make an asynchronous method call.

The following subsections cover the stages of preparing for the method call, invoking the call to one of the worker applications, and receiving the results from the asynchronous method call.

Preparing for an Asynchronous Method Call

The asynchronous method call is initiated by using a JadeMethodContext instance. The first step is to create such an instance and initialize it in readiness for invoking the asynchronous method call.

The following code fragment shows a transient instance of JadeMethodContext being created and initialized with the name of the worker application that will be used to process the request.

```plaintext
vars
  context : JadeMethodContext;
begin
  // Create the JadeMethodContext instance
```
create context transient;
// Mandatory - to name the worker application that will be used
context.workerAppname := "Server";
// Optional - to distinguish between multiple context objects
context.tag := 7;
...

Invoking the Asynchronous Method Call

The invoke method of the JadeMethodContext class submits the request for the method to be executed. The requests are handled by a JADE message queue and the method is executed by the first idle worker application.

At least one copy of the worker application should be running.

The method signature of the invoke method has a formal target parameter, which is a placeholder for the receiver object reference, the called method, and the list of parameters (if any) for the called method.

The following code fragment shows the use of the invoke method to make an asynchronous call to execute the getHistory method on an instance of the Customer class.

```java
vars
  context : JadeMethodContext;
  cust : Customer;
  date : Date;
begin
  ...
  // The context is supplied with the necessary information
  // for the asynchronous method call and the call is initiated
  context.invoke(cust, getHistory, date);
  // The asynchronous method call is now executing in a worker application
  // or waiting in a queue to be executed
  ...
```

The method signature of the getHistory method in the Customer class is as follows.

```java
Customer::getHistory(startDate: Date): String;
```

Timeout

A request that is reported to the caller as timed out completes processing but the response is discarded.

Waiting for Asynchronous Method Calls to Complete

When the asynchronous method call is executing in the worker application, the application that originated the request continues to execute instructions following the invocation of the asynchronous method call.

The Process class waitForMethods method enables an application to wait for the completion of a number of asynchronous method calls and obtain the results of the calls without having to return to an idle state. It is sometimes necessary to wait for the asynchronous calls to complete before other actions are taken.

A method context object whose state property is not State_Processing is ignored; that is, if it has not yet been invoked or has already completed. The state of a method context object is set to State_Completed when the context is returned as the result of the waitForMethods method.

A maximum of 64 processing method context objects can be passed to the waitForMethods method.
The following diagram shows a period of parallel processing when the asynchronous method call is executing at the same time as the method in the originating application. The originating application then calls the `waitForMethods` method, which suspends processing until the asynchronous method call completes (or times out). At that point, the originating application resumes processing.

The following code fragment shows the use of the `waitForMethods` method in this simple scenario.

```java
vars
custom: JadeMethodContext;
cust: Customer;
date: Date;
completedContext: JadeMethodContext;
begin
... // One asynchronous method call is made
context.invoke(cust, getHistory, date);
... // Wait for the asynchronous method call to complete
completedContext := process.waitForMethods(context);
write completedContext = context; // Outputs true
...
```

In the following diagram, the originating application makes two asynchronous method calls. The originating application then makes repeated calls to the `waitForMethods` method to receive information from the worker applications as the asynchronous method calls complete.

The first time around the loop, the `waitForMethods` method immediately returns `context2`, as it has already completed. The second time around the loop, it suspends processing until `context1` is completed. The loop is then terminated, because the `waitForMethods` method returns a null reference, indicating that all asynchronous method calls have completed.

The following code fragment shows the use of the `waitForMethods` method in this more-complicated scenario.

```java
vars
custom1, custom2: JadeMethodContext;
cust1, cust2: Customer;
date1, date2: Date;
completedContext: JadeMethodContext;
begin
... // Two asynchronous method calls are made
context1.invoke(cust1, getHistory, date1);
context2.invoke(cust2, getHistory, date2);
... completedContext := process.waitForMethods(context1, context2);
// Loop through contexts as they finish
```
while completedContext <> null do
  write completedContext;
  completedContext := process.waitForMethods(context1, context2);
endwhile;
...

Obtaining Results from an Asynchronous Method Call

The `waitForMethods` method on the `Process` class returns a `JadeMethodContext` instance for an asynchronous method call that has completed or timed out. This object provides methods to report on:

- Whether the call completed, timed out, or failed
- The error information (if it failed)
- The return value from the call (if there was one)

The following code fragment shows the result of an asynchronous method call being examined.

```java
vars
  context: JadeMethodContext;
  cust: Customer;
  date: Date;
  completedContext: JadeMethodContext;
begin
  ...
  // Obtain context of last asynchronous call to complete
  completedContext := process.waitForMethods(context);
  // Check how things went and obtain the result
  if completedContext = null then
    write "All calls completed";
  elseif completedContext.getErrorNumber <> 0 then
    write "Error info: " & completedContext.getErrorText;
  elseif completedContext.isTimedOut then
    write "Call timed out";
  else
    write "Return value: " & completedContext.getReturnValue.String;
  endif;
  ...
```

You can use the `getTimestamps` method to return the information listed in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>invokeTS</code></td>
<td>The timestamp when the asynchronous method call was invoked using the</td>
</tr>
<tr>
<td></td>
<td><code>JadeMethodContext</code> instance</td>
</tr>
<tr>
<td><code>beginTS</code></td>
<td>The timestamp value when the worker application began processing the</td>
</tr>
<tr>
<td></td>
<td>asynchronous method call</td>
</tr>
<tr>
<td><code>finishTS</code></td>
<td>The timestamp value when the worker application completed processing the</td>
</tr>
<tr>
<td></td>
<td>asynchronous method call</td>
</tr>
<tr>
<td><code>harvestTS</code></td>
<td>The timestamp when the receiver was returned by the <code>waitForMethods</code> method</td>
</tr>
<tr>
<td></td>
<td>and reported as complete</td>
</tr>
<tr>
<td><code>qdepth</code></td>
<td>The number of requests already queued when the asynchronous method call</td>
</tr>
<tr>
<td></td>
<td>request was added to the queue</td>
</tr>
</tbody>
</table>
If the `getTimestamps` method is called before the `invoke` method is called, none of the output parameters returns a value.

If the `getTimestamps` method is called after the `invoke` method is called but before the `waitForMethods` method is called, only the `invokeTS` and `qdepth` parameters return a value.

If the `getTimestamps` method is called after the `waitForMethods` method is called, all output parameters return a value.

**Example of Using Asynchronous Method Calls in a Text Search**

The example in this section searches for a text string (`targetText`) in the method source of a set of classes (`classList`). The example is broken up into a number of work units, to allow the searching of the classes to be carried out by a number of worker applications in parallel. Each work unit is a group of 10 classes from the total list of classes in which the search is performed.

Each work unit uses the `searchClassesMethods` method in the `Application` class. The code that demonstrates how the search is implemented is not relevant to the example and is not given.

The signature of the `searchClassesMethods` method is as follows.

```java
searchClassesMethods(classList: XClassSet;
                      targetText: String;
                      caseSensitive: Boolean;
                      matches: XMethodSet input): Integer;
```

The `classList` parameter contains the set of 10 classes for the work unit, and is a collection class with a membership of `Class` instances. The method searches the local methods of each listed class, adding each method that has the requested text to the `matches` collection (which is a shared transient instance of a collection type with a membership of `Method` instances), and returns the count of matches found.

The signature of the method in the originating application that initiates the search is as follows.

```java
parallelSearch10(targetText: String;
                 caseSensitive: Boolean;
                 classList: ClassColl;
                 parallelRequests: Integer);
```

The `target` parameter is the search text string and the `classList` parameter is the list of classes whose methods are to be searched. The `caseSensitive` parameter specifies whether the search is case-sensitive.

The `parallelRequests` parameter is the number of parallel work units initiated. In this example, the work units are implemented by instances of a subclass of the `JadeMethodContext` class called `CallContext`. The subclass has a non-exclusive `matchingMethods` reference, which is of type `XMethodSet`, and a non-exclusive shared transient `classList` reference, which is of type `XClassSet`.

**Note** The number of parallel work units initiated, which is the number of `CallContext` objects, differs from the number of worker applications that are running. Work units are queued until worker applications become available.

The code for the `parallelSearch10` method is as follows.

```java
var
  cls: Class;
  clsIndex: Integer;
  totalClasses: Integer;
```
totalMatches: Integer;
doneClasses: Integer;
emptyClasses: Integer;
matchCount: Integer;
meth: Method;
errnum: Integer;
countMeths: Integer;
inx: Integer;
iter: Iterator;
running: Boolean;
jmtthCntxX: JadeMethodContext;
callCntxX: CallContext;
allCallContexts: CallContextArray;

begin
  totalClasses :=classList.size;
totalMatches := 0;
emptyClasses := 0;
// Create a call context for each parallel request
create allCallContexts transient;
beginTransientTransaction;
foreach inx in 1 to parallelRequests do
  create callCntxX transient;
  allCallContexts.add(callCntxX);
callCntxX.tag := inx;
callCntxX.workerAppName := "SearchWorker";
callCntxX.classList sharedTransient;
callCntxX.matchingMethods sharedTransient;
endforeach;
commitTransientTransaction;
// Start the initial requests
iter :=classList.createIterator;
foreach callCntxX in allCallContexts do
  beginTransientTransaction;
callCntxX.classList.clear;
clsIndex := 0;
  while iter.next(cls) do
    countMeths := cls.countLocalMethods;
    if countMeths = 0 then
      doneClasses := doneClasses + 1;
      emptyClasses := emptyClasses + 1;
      continue;
    endif;
    callCntxX.classList.add(cls);
    clsIndex := clsIndex + 1;
    if clsIndex = 10 then
      break;
    endif;
  endwhile;
commitTransientTransaction;
if clsIndex > 0 then
  callCntxX.initialize;
callCntxX.classList := clsIndex;
callCntxX.invoke(app, searchClassesMethods, callCntxX.classList,
                  targetText, caseSensitive, callCntxX.matchingMethods);
endif;
endforeach;
running := true;
while running do
    // wait for a request to complete
    jmthCntxX := process.waitForMethods(allCallContexts);
    if jmthCntxX = null then //No active contexts
        running := false;
    else
        callCntxX := jmthCntxX.CallContext;
        matchCount := 0;
        if callCntxX.isTimedOut then
            errnum := 9997;
        else
            errnum := callCntxX.getErrorNumber;
            if errnum = 0 then
                matchCount := callCntxX.getReturnValue.Integer;
            endif;
        endif;
        if matchCount > 0 then
            foreach meth in callCntxX.matchingMethods do
                write "MATCH " & meth.schemaType.schema.name & "::" &
                    meth.schemaType.name & "::" & meth.name;
            endforeach;
            totalMatches := totalMatches + matchCount;
        endif;
        doneClasses := doneClasses + callCntxX.classCount;
    // Start another request
    clsIndex := 0;
    beginTransientTransaction;
    callCntxX.classList.clear;
    while iter.next(cls) do
        countMeths := cls.countLocalMethods;
        if countMeths = 0 then
            doneClasses := doneClasses + 1;
            emptyClasses := emptyClasses + 1;
            continue;
        endif;
        callCntxX.classList.add(cls);
        clsIndex := clsIndex + 1;
        if clsIndex = 10 then
            break;
        endif;
    endwhile;
    commitTransientTransaction;
    if clsIndex > 0 then
        callCntxX.initialize;
        callCntxX.classCount := clsIndex;
        callCntxX.invoke(app, searchClassesMethods,
                         callCntxX.classList, targetText,
                         caseSensitive, callCntxX.matchingMethods);
    endif;
    endif;
endwhile;
write "Searched " & doneClasses.String & " of " & totalClasses.String &
" classes, skipping " & emptyClasses.String & CrLf &
totalMatches.String & " matches";
epilog
beginTransientTransaction;
foreach callCntxX in allCallContexts do
    delete callCntxX.classList;
    delete callCntxX.matchingMethods;
    delete callCntxX;
endforeach;
commitTransientTransaction;
delete allCallContexts;
end;
Chapter 17  Using the JADE Testing Framework

This chapter contains the following topics.

- **Overview**

- **JADE Unit Testing Framework**
  - Writing Unit Tests
    - Generating a Web Service Consumer Unit Test Class and Stub Methods
  - Running Unit Tests
    - Running Unit Tests from the JADE Development Environment
    - Using the Unit Test Runner Form
    - Running Unit Tests In Batch Mode
    - Running Batch Mode Unit Tests that Require GUI Components
    - Running Unit Tests from an Application
  - Listening for Unit Test Results
  - Example of Using the JADE Unit Test Framework

- **Code Coverage**
  - Collecting Code Coverage Results
  - Viewing Code Coverage Results
  - Code Coverage Results Browser
  - Using Code Coverage Menus

**Overview**

Unit testing is a simple but effective approach to improving the quality and reliability of a system. The key idea is to develop tests for a unit of code (which may be a fragment, a method or a module) as the code is being written. Development teams can employ this approach to ensure that quality is built into a project from the early stages.

The JADE unit testing framework enables unit tests to be executed automatically and for the test results to be captured. For details about unit testing, see "JADE Unit Testing", later in this chapter.

The goal of unit testing is to isolate each part of the program and show that the individual parts are correct. A unit test provides a strict, written contract that the piece of code must satisfy. Unit tests provide the following benefits.

- Enabling refactoring without regression (that is, ensuring the module still works correctly)
- Eliminating uncertainty in the units
- Enabling a **bottom-up** testing style approach
- Documenting the functionality provided by a unit and how to use it
Having developed a number of unit tests, the question of how comprehensively they test the system arises. Code coverage is a measure used in software testing to describe the degree to which the methods in a system have been tested. It is a useful measure to assure the quality of a set of tests, as opposed to directly reflecting the quality of the system under test.

Code coverage can help testers and developers to:

- Discover methods and blocks of code that are not exercised by a set of tests
- Create tests that increase code coverage
- Quantify the overall code coverage of a system, which is one measure of quality

For details about code coverage, see "Code Coverage", later in this chapter.

**JADE Unit Testing**

The JADE unit testing framework is part of the RootSchema and consists of classes and an interface, which are documented in Volume 2 of the JADE Encyclopaedia of Classes.

<table>
<thead>
<tr>
<th>Class or Interface</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JadeTestCase</td>
<td>Superclass for all user-written unit test classes</td>
</tr>
<tr>
<td>JadeTestListenerIF</td>
<td>Interface providing callback methods on the progress and results of unit testing</td>
</tr>
<tr>
<td>JadeTestRunner</td>
<td>Class for running user-written unit tests</td>
</tr>
</tbody>
</table>

Unit test methods are written in subclasses of **JadeTestCase** from which they inherit the assertion methods that determine whether a unit test fails or succeeds. A unit test run can be structured with methods to set up data in a known state (the test fixture) before a unit test starts and to tear down that data when the unit test has completed.

The results of a unit test run can be received by an object that implements the **JadeTestListenerIF** interface (the listener object). The way the results are captured, displayed, and analyzed depends on how the listener object implements the interface methods.

The JADE development environment enables you to execute a selected unit test method or all unit test methods for a selected class by pressing the F9 key. For details, see "Running a Unit Test", in Chapter 4 of the **JADE Development Environment User's Guide**.

**Note** Visual Studio uses the F9 key to set breakpoints, by default, and uses the F5 key to run and continue debug execution. JADE uses the F5 key to set breakpoints and the F9 key to run and continue debug execution. Switching between the two development environments can lead to confusion and frustration when using the F5 and F9 keys.

You can swap the F9 and F5 accelerator key bindings, by checking the **Swap F5 (Toggle Breakpoint) and F9 (Execute/Continue) accelerators** check box on the **Shortcut Keys** sheet on the Preferences dialog. This check box is unchecked by default (that is, false). For details, see "Maintaining Shortcut Keys", in Chapter 2 of the JADE Development Environment User's Guide.

**Writing Unit Tests**

The framework class **JadeTestCase** provides assertion methods that you can use in your unit test methods. The example used in the following sections to illustrate how unit tests are written is a system that enables a user to carry out arithmetical calculations such as adding, subtracting, and so on. The example system could also convert metric values to imperial, and the reverse.
The system would have forms (for example, CalculatorForm and ConvertorForm) to provide the GUI interface, but the arithmetical and conversion functionality would be encapsulated in separate classes Calculator and Convertor, as shown in the following image.

Whilst the forms do have to be tested for the GUI functionality, the unit tests focus more on the underlying arithmetical and conversion calculations carried out by the functionality classes. For example, a unit test could be written to test the following add method in the Calculator class.

```java
add(n : Integer) updating;
begin
  // 'result' is an attribute in the Calculator class
  result := result + n;
end;
```

Typically there is one unit test class for each functionality class, as shown in the following image.

It is possible, but not usual, to combine the unit test classes, as shown in the following image.

It is also possible, but not usual, to split testing between two unit test classes, as shown in the following image.

Unit test classes are created as subclasses of the JadeTestCase class. The unit test methods of TestCalculator are designed to put the Calculator class through its paces.

A unit test method asserts the truth of a statement such as the existence or non-existence of an object, the equality of two values, and so on. The methods for making these assertions are provided in the JadeTestCase class. A test fails if it does not perform as expected.
An exception can occur when running a test. Under certain circumstances, the exception is expected and a unit test could be written to confirm that it is raised. Expected exceptions are registered using the `expectedException` method provided by the `JadeTestCase` class.

When writing a unit test method, you often have to create test data so that the test is performed for a system in a known state. In addition, you may need to remove the test data when the unit test method is finished. Each unit test method could be responsible for creating and removing its own test data, but the test framework enables you to create separate methods to do this.

A method in a unit test class can be tagged with a method option that indicates its purpose and therefore at what stage in a test run it is executed.

Methods with `unitTestBeforeAll` and `unitTestAfterAll` method options can be defined only on the base `JadeTestCase` class, and there can be one instance only of these method options in a schema.

If any number of tests are run from the schema, the `unitTestBeforeAll` and `unitTestAfterAll` method options are executed once only.

If multiple `unitTestBeforeAll` and `unitTestAfterAll` method options are defined on methods of the `JadeTestCase` class in a schema, the first instance of the method option is used and a warning is logged, stating that all other instances have been ignored.

If `unitTestBeforeAll` and `unitTestAfterAll` method options are defined on methods in subclasses of the `JadeTestCase` class, they are ignored and a warning is logged, stating they have been ignored.

The following examples show method options available for methods in a `JadeTestCase` subclass.

- The `unitTestBeforeAll` method option identifies the method that is run once before any classes are tested.

  ```
  unitTestSubclass1_beforeAll() unitTestBeforeAll;
  begin
      MethodCallCounts@deleteInstance();
      MethodCallCounts@incrementValue(method.name);
  end;
  ```

  A method with the `unitTestBeforeAll` option specified can perform any one-off global initialization for all tests; for example, package initialization and creating test data.

- The `unitTestBeforeClass` method option identifies the method that is run before running a group of unit test methods for a unit test class.

  ```
  turnonCalculator() unitTestBeforeClass, updating;
  begin
      create calculator transient;
      calculator.switchOn;
  end;
  ```

- The `unitTestBefore` method option identifies the method that is run before running each unit test method in a unit test class.

  ```
  clearTestBefore() unitTestBefore;
  begin
      calculator.clear;
  end;
  ```

- The `unitTest` method option identifies a unit test method that is executed.

  ```
  add() unitTest;
  begin
      calculator.add(1);
  ```
calculator.add(1);
assertEquals(2, calculator.getResult());
end;

- The **unitTestAfterAll** method option identifies the method that is run once after all classes have been tested.

  ```java
  unitTestSubclass1_afterAll() unitTestAfterAll;
  begin
    MethodCallCounts@incrementValue(method.name);
  end;
  
  A method with the **unitTestAfterAll** option specified can perform any one-off global clean up for all tests; for example, package finalization and deleting test data.

- The **unitTestAfterClass** method option identifies the method that is run once after running a group of unit test methods for a unit test class.

  ```java
  turnonCalculator() unitTestAfterClass;
  begin
    MethodCallCounts@incrementValue(method.name);
  end;
  ```

The method options available for methods in a unit test class are listed in the following table.

<table>
<thead>
<tr>
<th>Method Option</th>
<th>This method is ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>unitTestBeforeAll</td>
<td>Run once before any classes are tested. Only one method in the class can have this method option.</td>
</tr>
<tr>
<td>unitTestBeforeClass</td>
<td>Run once before the class is tested. Only one method in the class can have this method option.</td>
</tr>
<tr>
<td>unitTestAfterClass</td>
<td>Run once after the class is tested. Only one method in the class can have this method option.</td>
</tr>
<tr>
<td>unitTestAfterAll</td>
<td>Run once after all classes have been tested. Only one method in the class can have this method option.</td>
</tr>
<tr>
<td>unitTestBefore</td>
<td>Run before each unit test method of the class is executed. Only one method in a <strong>JadeTestCase</strong> subclass should have this option.</td>
</tr>
<tr>
<td>unitTestAfter</td>
<td>Run after each unit test method of the class is executed. Only one method in a <strong>JadeTestCase</strong> subclass should have this option.</td>
</tr>
<tr>
<td>unitTest</td>
<td>A unit test method. It cannot have parameters or return a result. The test fails if an <strong>assertion</strong> method fails or an exception is raised.</td>
</tr>
<tr>
<td>unitTestIgnore</td>
<td>A unit test method that is ignored when the tests are run. A possible reason for not running the test is that the unit test method is incomplete or the underlying functionality to be tested is incomplete.</td>
</tr>
</tbody>
</table>

---

### Generating a Web Service Consumer Unit Test Class and Stub Methods

You can generate a set of stub methods that can be used for Web service consumer unit testing. If the schema is versioned and you are working in the latest version, the existing **JadeTestCase** subclass is versioned if it is unversioned.
In the description in this section, the **WSD_ErewhonInvestmentsService** sample Web service consumer is used, which exposes the methods shown in the following image.

To generate stub methods for your Web service consumer

1. Select the **Generate Web Consumer Test Case** command from the Classes menu in the Class Browser. (This command is enabled only when the selected class is a subclass of the **JadeWebServiceConsumer** class.)
The Generate Test Case dialog, shown in the following image, is then displayed.

2. In the Test Name text box, change the unit test class, if required. The name of the JadeTestCase subclass that is created defaults to the Web service consumer subclass name (WSD_ErewhonInvestmentsService, in the example in this section) with the Test_ prefix. The Generate Test Case dialog lists all methods in the JadeWebServiceConsumer subclass that have the webService method option defined.

3. In the table of methods defined in the Web service consumer subclass, check the check box in the Select column of each method for which you want a test case generated. Conversely, if you want to select all of the methods, check the Select All Methods check box. (Unchecking this check box does not affect the list of selected methods.)

4. Check the Generate Parameter Defaults check box if you want to generate stub code for the method parameters.

5. Check the Assert all tests as incomplete check box if you want to add an assert to the method code to say that the test is not complete.

6. Check the Generate Expected Defaults check box if you want to generate stub code for the expected return value.

7. Click the OK button. Alternatively, click the Cancel button to abandon your selections. If you click the OK button and you have not selected any methods, No methods were selected - Continue? is displayed in a message box. Click No if you want to cancel the process and return to the dialog or click Yes to continue with the generate process.
For the example in this section, the name of the test has been left as `Test_WSD_ErewhonInvestmentsService` and all of the check boxes have been checked.

When you are generating a test case for the same `JadeWebServiceConsumer` subclass into an existing `JadeTestCase` class, the generate process:

- Creates the required Web service consumer class property reference on the class, if it does not exist.
- Creates any methods selected in the displayed table that do not exist in the specified `JadeTest` subclass.
- Creates the `create` method, if it does not exist.
- Warns you if the `create` method already exists and it does not contain the `create property-name web-service-consumer-class-name;` statement. You must manually add that statement to the `create` method.
- Ignores selected methods when a method of that name already exists in the `JadeTestCase` subclass.
- Displays a message box on completion of the generate process, warning you if selected methods are ignored because a method of that name already exists.

When you click the OK button, the `JadeTestCase` subclass, shown in the following image, is then generated.
In addition to the stub methods generated for each of the methods defined in the Web service, a create method and a wSD_ErewhonInvestmentsService property that references the Web service are also generated.

The value of the wSD_ErewhonInvestmentsService property is initialized in the create method to the Web service from which the test was generated.

The code that was generated for the getClient method is shown in the following image.

```java
getClient() { unitTest;

    // Create WSD_Client instance and initialise primitive property values
    create _get clientIdResult transient;
    _get clientIdResult.address1 := "";
    _get clientIdResult.address2 := "";
    _get clientIdResult.address3 := "";
    _get clientIdResult.email := "";
    _get clientIdResult.fax := "";
    _get clientIdResult.name := "";
    _get clientIdResult.phone := "";
    _get clientIdResult.website := "";

    // Initialise parameters to the appropriate value
    create inputParameters transient;
    inputParameters.clientName := "";

    // Call the web service method
    actualValue := wSD_ErewhonInvestmentsService.getClient(inputParameters);

    // Test the returned value and the expected value
    assertEquals(expectedValue, actualValue);
    assert("Verify the correctness of this test method.");
}
```

In this code:

- Lines 12 through 24 represent the expected defaults and are generated when the Generate Expected Defaults check box is checked.
- Lines 27 and 28 represent the parameter default values that are generated when the Generate Parameter Defaults check box is checked.
- Line 31 is the call to the Web service.
- Line 33 is the default assert.

When the Assert all tests as Incomplete check box is checked and a method is in error, an assert line
advices you that the correctness of the test method requires verification.

**Note**  This assert serves only as a reminder to you that the method needs to be changed.

As this is a stub method only, running this test without any modifications is not likely to work; for example, pressing F9 on the method in the above example to run the method results in a failed assert. The method test fails because the expected value does not match the return value.

Running a test when there is a web service fault such as an HTTP error results in the test failing with an unexpected error, as shown in the following image.

![Unit Test Runner Image]

If an assert advises you that the correctness of a test method requires verification, that assert always fails.
When the test has been coded correctly, the assert should be removed. For example, if you want the test to check that the **name** property in the **Client** object returned by the Web service is the same as the requested name, the following image shows the required method modifications.

```java
1 getclient() unitTest;
2
3 // Generated on 14 February 2018 by Wilbur
4
5 var
6     actualValue: WSD_GetClientResponse;
7     expectedValue: WSD_GetClientResponse;
8     inputParameters : WSD_GetClient;
9     _getClientResult : WSD_Client;
10 begin
11     // create the expected return value and initialise properties
12     create expectedValue transient;
13
14     // Create WSD_Client instance and initialise primitive property values
15     create _getClientResult transient;
16     expectedValue._getClientResult := _getClientResult;
17     _getClientResult.address1 := "";
18     _getClientResult.address2 := "";
19     _getClientResult.address3 := "";
20     _getClientResult.email := "";
21     _getClientResult.fax := "";
22     _getClientResult.name := "Sidney";
23     _getClientResult.phone := "";
24     _getClientResult.webSite := "";
25
26     // Initialise parameters to the appropriate value
27     create inputParameters transient;
28     inputParameters.clientName := "Sidney";
29
30     // Call the web service method
31     actualValue := WSD_ErwhonInvestmentsService.getClient(inputParameters);
32     // Test the returned value and the expected value
33     assertEquals(expectedValue, actualValue);
34 end;
```

Note that the last assert has been removed and that client name Sidney has been set up. The result of running this modified test is that the test has a status of **passed**, highlighted with a green background, and the test in the **Select Tests** pane at the left of the Unit Test Runner form displays a check mark symbol.

Similarly, you can set up other expected values and compare these to the return values.

For details about the Unit Test Runner form, see the "Using the Unit Test Runner Form", later in this chapter.

**Note**  It makes sense to compare primitive type values only, as returned object instances will not match because they will always be different (transient) object instances.
Running Unit Tests

You can run unit test methods from:

- The Class Browser, by selecting a unit test class or method and then pressing the F9 key or by selecting the Unit Test command from the Jade menu. (To run unit tests in debug mode, press Shift+F9 or select the Unit Test Debug command from the Jade menu.)

- The Schema Browser, by selecting a schema that has a unit test class and then pressing the F9 key or by selecting the Unit Test command from the Jade menu. (To run unit tests in debug mode, press Shift+F9 or select the Unit Test Debug command from the Jade menu.)

- A batch of tests, using a RootSchema application provided for that purpose.

- Code, by using the runTests method of the JadeTestRunner class or by displaying the Unit Test Runner form.

Running unit tests from code is considered the exceptional case. You would consider doing this only if you want more control over the presentation of the test results.

**Notes**  The name of the JADE unit testing application is JadeUnitTest and that of the batch version is JadeUnitTestRun. It may be necessary to change your getAndValidateUser method to enable the unit testing applications to run.

JADE does not initialize any packages for RootSchema applications running in user schemas, including unit tests run using the JADE Unit Test framework in the RootSchema. If you require this initialization, you must initialize packages in your user test code (using the Process class initializePackages and finalizePackages methods). Initialization could be performed in the unitTestBeforeAll method and finalization in the unitTestAfterAll method.

When the unit tests finish running, the final failure count reflects the following:

- If a class instance create fails, the first class method executed will be counted as failed.

- If a unitTestBeforeClass method fails, the first class method executed will be counted as failed.

- If a unitTestAfterClass method fails, the last class method executed will be counted as failed.

- Any failure associated with a method will count as a failure only in the final failure statistics, whether it is a failure resulting from any of the previous items in this list or from a failure in the unitTestBefore, unitTestAfter, or unitTest method.

Running Unit Tests from the JADE Development Environment

You can run unit tests from the JADE development environment, by selecting a unit test method or class and then pressing the F9 key. Alternatively, select the Unit Test command from the Jade menu. For details, see "Running a Unit Test", in Chapter 4 of the JADE Development Environment User's Guide.

The Unit Test Runner form is then displayed, with the results for the unit tests are displayed in the Results list box. For details about the Unit Test Runner form, see the "Using the Unit Test Runner Form", in the following section.

The following table lists unit tests that are run for an item selected in the Class Browser.

<table>
<thead>
<tr>
<th>Selected item</th>
<th>Unit tests that are run</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit test method</td>
<td>The selected method is executed like a JadeScript method</td>
</tr>
<tr>
<td>Unit test class (a subclass of JadeTestCase)</td>
<td>All unit test methods in the selected class and its subclasses</td>
</tr>
</tbody>
</table>
Using the Unit Test Runner Form

You can run unit tests by selecting the Unit Test command from the Jade menu in the JADE development environment. For details, see "Running a Unit Test", in Chapter 4 of the JADE Development Environment User’s Guide.

The Unit Test Runner form is then displayed, with unit test classes for the current schema and superschemas listed in the Select Tests list box. This list box includes status icons for all selected tests. These indicate whether a test has passed (green tick), failed (red cross), or has not been run (blue question mark).

The Unit Test Runner form can also be displayed from code, as shown in the following example.

```pascal
vars
dlg : JadeTestDialog;
begin
    create dlg;
    dlg.show;
end;
```

Select the class or classes whose unit test methods are to be executed and then click the Run button.

The results of the tests are displayed in the Results pane, as shown the following image.

To run tests from the Unit Test Runner form, select the class or test in the Select Tests pane and then click the Run button.
To copy the unit test results to the clipboard, right-click in the Results pane.

**Performing File Menu Command Actions**

The Unit Test Runner form File menu provides the commands listed in the following table.

<table>
<thead>
<tr>
<th>Command</th>
<th>For details, see…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug on Assert</td>
<td>Displaying the Call Stack for Failed Methods</td>
</tr>
<tr>
<td>Debug on Exception</td>
<td>Displaying the Call Stack for Exceptions</td>
</tr>
<tr>
<td>Debug on Unexpected Exception</td>
<td>Displaying the Call Stack for Unexpected Exceptions</td>
</tr>
<tr>
<td>Code Coverage</td>
<td>Enabling Code Coverage</td>
</tr>
<tr>
<td>Report Code Coverage</td>
<td>Reporting Code Coverage Results</td>
</tr>
<tr>
<td>View Code Coverage</td>
<td>Viewing Code Coverage Results</td>
</tr>
<tr>
<td>Profile</td>
<td>Starting the Collection of Unit Test Results</td>
</tr>
<tr>
<td>Report Profile</td>
<td>Outputting Unit Test Results to a File</td>
</tr>
<tr>
<td>Exit</td>
<td>Exiting from the Unit Test Runner Form</td>
</tr>
</tbody>
</table>

For details, see the following sections.

**Increasing the Font Size**

- **To increase the size of the displayed font, perform one of the following actions**
  - Select the **Font Bigger** command from the View menu.
  - Press Ctrl+1.

  The font size is then doubled (that is, increased by a factor of 100 percent).

  You can repeat this action once more if you want the displayed font to be 200 percent larger than the default font (that is, this command is disabled when you have selected it twice in succession).

**Note**  The `UIFont` parameter in the [JadeUnitTestRunnerUI] section of the JADE initialization file specifies the size in points of the font used to display information in the Unit Test Runner form. The value can be 10 (the default), 11, or 12.

**Decreasing the Font Size**

- **To decrease the increased size of the displayed font, perform one of the following actions**
  - Select the **Font Smaller** command from the View menu.

    This command is disabled when the text in the Unit Test Runner form is the default font size.
  - Press Ctrl+0.

    The font size is then halved (that is, decreased by a factor of 100 percent).

    You can repeat this action once more if you want the displayed font to be 200 percent smaller than the maximum increased font size and returned to the default font size (that is, this command is disabled when you have selected it twice in succession).
Note: The `UIFontSize` parameter in the `[JadeUnitTestRunnerUI]` section of the JADE initialization file specifies the size in points of the font used to display information in the Unit Test Runner form. The value can be **10** (the default), **11**, or **12**.

Displaying the Call Stack for Failed Methods

To display the call stack for failed methods

- Select the **Debug on Assert** command from the Unit Test Runner form File menu.

This command determines whether the unit test is paused and the call stack is displayed when a unit test fails on assert.

A check mark is then displayed at the left of the menu item, indicating that the call stack is displayed for the failed test during a unit test run. When the call stack window is closed, the unit tests resume.

To toggle the display of the call stack for failed messages, select the **Debug on Assert** command again, to remove the check mark and run all of the unit tests without pausing.

You can also double-click on a failed method in the **Results** pane, to display call stack information, as shown in the following image.

![Call Stack Image]

The value of this command is saved in the **DebugOnAssert** parameter in the `[JadeUnitTestRunnerUI]` section of the JADE initialization file. This parameter is read and used when the application is next initialized.

Displaying the Call Stack for Exceptions

To display the call stack for exceptions

- Select the **Debug on Exception** command from the Unit Test Runner form File menu.

This command determines whether the unit test is paused and the call stack is displayed when the unit test encounters an exception. A check mark is then displayed at the left of the menu item, indicating that the call stack is displayed for the failed test during a unit test run. When the call stack window is closed, the unit tests resume.
To toggle the display of the call stack for failed messages, select the Debug on Exception command again, to remove the check mark and run all of the unit tests without pausing.

You can also double-click on a method that failed with an exception in the Results pane, to display call stack information.

The value of this command is saved in the DebugOnException parameter in the [JadeUnitTestRunnerUI] section of the JADE initialization file. This parameter is read and used when the application is next initialized.

Displaying the Call Stack for Unexpected Exceptions

To display the call stack for unexpected exceptions

- Select the Debug on Unexpected Exception command from the Unit Test Runner form File menu.

This command determines whether the unit test is paused and the call stack is displayed when the unit test encounters an unexpected exception. This enables you to debug exceptions other than any that the test has registered with the JadeTestCase class expectedException method.

A check mark is then displayed at the left of the menu item, indicating that the call stack is displayed for the failed test during a unit test run. When the call stack window is closed, the unit tests resume.

To toggle the display of the call stack for failed messages, select the Debug on Unexpected Exception command again, to remove the check mark and run all of the unit tests without pausing.

You can also double-click on a method that failed with an exception in the Results pane, to display call stack information.

The value of this command is saved in the DebugOnUnexpectedException parameter in the [JadeUnitTestRunnerUI] section of the JADE initialization file. This parameter is read and used when the application is next initialized.

Enabling Code Coverage

To enable code coverage on unit tests

- Select the Code Coverage command from the Unit Test Runner form File menu.

This command determines whether code coverage is recorded when running unit tests. Using code coverage enables you to determine which blocks of code have executed in your unit tests.

The results of code coverage can be logged in a report or viewed directly, using the Code Coverage Results Browser.

The value of this command is saved in the CodeCoverage parameter in the [JadeUnitTestRunnerUI] section of the JADE initialization file. This parameter is read and used when the application is next initialized.

Reporting Code Coverage Results

To report code coverage results for unit tests

- Select the Report Code Coverage command from the Unit Test Runner form File menu.

The code coverage results are logged to a code coverage file, which defaults to the jadeunittest_timestamp.ccd file, where the timestamp value is in the YYYYMMDD_hhmss format (for example, jadeunittest_20161021_085955.ccd).
The report is output to the directory specified by the `CodeCoverageDirectory` parameter in the [JadeProfiler] section of the JADE initialization file, which defaults to `logs/CodeCoverage` (that is, the `CodeCoverage` subdirectory of `logs`). For details, see "JADE Initialization File" (that is, the JADE Initialization File Reference).

**Viewing Code Coverage Results**

- **To view code coverage results for unit tests**
  - Select the View Code Coverage command from the Unit Test Runner form File menu.

The Code Coverage Results Browser is then displayed, showing the results of the code coverage. For details, see "Code Coverage Results Browser", later in this chapter.

**Starting the Collection of Unit Test Results**

The `Profile` command in the Unit Test Runner form File menu starts recording unit test results.

These unit test results are recorded until you output them to a file by selecting the `Report Profile` command, or you select the `Profile` command again, which unchecks (disables) the command.

- **To start the collection of until test results**
  - Select the `Profile` command from the Unit Test Runner form File menu.

The recording of the unit test results is then started and a checkmark symbol is displayed at the left of the `Profile` command.

**Outputting Unit Test Results to a File**

- **To output profiled unit test results to a file**
  - Select the `Report Profile` command from the Unit Test Runner form File menu.

The profile statistics are then output to the `JadeProfiler_application-name_yyyyMMdd_hhmmss.log` file in the path specified by the `ResultsFile` parameter in the [JadeProfiler] section of the JADE initialization file, if any. A new report file is generated for each call to the `JadeProfiler` class `report` method. If a path is not specified in this parameter, the file is located in the log file directory, specified by using the `LogDirectory` parameter in the [JadeLog] section of the JADE initialization file. (For details, see Chapter 1 of the JADE Initialization File Reference, "JADE Initialization File Reference").

You can view your profile output by using a text editor (for example, Notepad) or you can output it to a printer, if required. The statistics provided by the profile report enable you to optimize the code of your methods in the JADE development environment by analyzing the performance of your methods. For details about the statistics that are output, see "Reporting Profile Statistics", in Chapter 1 of the JADE Runtime Application Guide.

When you select the `Report Profile` command and there is an existing profile file, records are appended to existing records, indicated by start and finish times.

**Exiting from the Unit Test Runner Form**

- **To exit from the Unit Test Runner form**
  - Select the `Exit` command from the Unit Test Runner form File menu.

The Unit Test Runner form is then closed.
Performing View Menu Command Actions

The Unit Test Runner form View menu provides the commands listed in the following table.

<table>
<thead>
<tr>
<th>Command</th>
<th>For details, see…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Include Passed Tests in Results</td>
<td>Including Passed Tests in the Results Pane</td>
</tr>
<tr>
<td>Select Passed Tests</td>
<td>Selecting Passed Tests in the Select Tests Pane</td>
</tr>
<tr>
<td>Select Failed Tests</td>
<td>Selecting Failed Tests in the Select Tests Pane</td>
</tr>
<tr>
<td>Select Not Run Tests</td>
<td>Selecting Not Run Tests in the Select Tests Pane</td>
</tr>
<tr>
<td>Refresh</td>
<td>Refreshing the Unit Test Results</td>
</tr>
<tr>
<td>Font Bigger</td>
<td>Increasing the Font Size</td>
</tr>
<tr>
<td>Font Smaller</td>
<td>Decreasing the Font Size</td>
</tr>
</tbody>
</table>

For details, see the following sections.

Including Passed Tests in the Results Pane

To include passed tests in the Results pane of the Unit Test Runner form

- Select the Include Passed Tests in Results command from the Unit Test Runner form View menu.

This command determines whether the tests that are executed successfully are displayed in the Results pane. A check mark is then displayed at the left of the menu item, indicating that when the unit tests are run, the Results pane displays all test results; that is, tests that have passed, been skipped, or that failed.

To toggle the display of passed tests in the Results pane, select the Include Passed Tests in Results command again, to remove the check mark and display only tests that have been skipped or that failed.

The value of this command is saved in the IncludePassedTests parameter in the [JadeUnitTestRunnerUI] section of the JADE initialization file. This parameter is read and used when the application is next initialized.

Selecting Passed Tests in the Select Tests Pane

To select passed tests in the Select Tests pane of the Unit Test Runner form

- Select the Select Passed Tests command from the Unit Test Runner form View menu.

All passed tests (that is, tests with a green tick) are then selected in the Select Tests pane. Selected tests can then be run by clicking the Run button.

Selecting Failed Tests in the Select Tests Pane

To select failed tests in the Select Tests pane of the Unit Test Runner form

- Select the Select Failed Tests command from the Unit Test Runner form View menu.

All failed tests (that is, tests with a red cross) are then selected in the Select Tests pane. Selected tests can then be run by clicking the Run button.
Selecting Not Run Tests in the Select Tests Pane

To select not-run tests in the Select Tests pane of the Unit Test Runner form

- Select the Select Not Run Tests command from the Unit Test Runner form View menu.

All not-run tests (that is, tests with a blue question mark) are then selected in the Select Tests pane.

You can use this command to select unit tests that have been recompiled or created in the class since tests were last run in the Unit Test Runner. Selected tests can then be run by clicking the Run button.

Refreshing the Unit Test Results

To refresh the unit test results displayed in the Results pane of the Unit Test Runner form

- Select the Refresh command from the Unit Test Runner form View menu. Alternatively, press F5.

The results from the previous execution are cleared and the unit tests are then run again.

Running Unit Tests in Batch Mode

You can run the JadeUnitTestBatch application:

- As a standard client or a thin client from the JADE executable (jade.exe) command line, see "Running a JADE Client Application using jade.exe", in Chapter 1 of the JADE Runtime Application Guide.

The JadeUnitTestBatch application requires the elements within the startAppParameters and endAppParameters arguments of the jade.exe command line.

To run unit tests in batch mode as a non-GUI application using jade.exe, specify the following arguments in the command line.

```
jade path=path database-path
    schema=RootSchema
    ini=jade-initialization-file
    app=JadeUnitTestBatch
    startAppParameters
    control-file-name
    test-name
    [codeCoverage=true|false]
    [profile=true|false]
    [log-file-name]
    endAppParameters
```

The following example specifies the JADE executable (jade.exe) command line for the non-GUI JadeUnitTestBatch presentation client application.

```
jade path=c:\jade\system schema=TestUnitTestSchema ini=c:\jade\system\jade.ini
app=JadeUnitTestBatch appServer=MyAppServer appServerPort=1234
startAppParameters c:\jade\unittest.xml DailyTests codecoverage=true profile=true c:\jade\logs\unittest.log endAppParameters
```

For details about running the JadeUnitTestGuiNoForms application from jade.exe, see "Running Batch Mode Unit Tests that Require GUI Components", later in this chapter.

- Using the jadclient non-GUI client application to automate the batch running of unit tests, passing command line arguments after the startAppParameters argument to specify your testing and logging requirements.
For details about the jadclient non-GUI client application, see "Running a Non-GUI Client Application using jadclient", in Chapter 1 of the JADE Runtime Application Guide.

To run unit tests in batch mode as a non-GUI application using jadclient, specify the following arguments in the command line.

```
jadclient path=database-path
   ini=jade-initialization-file
   app=JadeUnitTestBatch
   schema=schema-name
   startAppParameters
   control-file-name
   test-name
   [profile=true|false]
   [log-file-name]
   [codeCoverage=true|false]
```

The following is an example of the command line that runs a non-GUI client application to run the DailyTests unit test, which is defined in the c:\jade\unittest.xml XML file.

```
jadclient path=c:\jade\system ini=c:\jade\jade.ini server=multiUser
   schema=RootSchema app=JadeUnitTestBatch startAppParameters c:\jade\unittest.xml
   DailyTests c:\jade\logs\unittest.log
```

The arguments following the startAppParameters argument are listed in the following table.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>control-file-name</td>
<td>Name of the XML control file that defines groups of tests that can be run.</td>
</tr>
<tr>
<td>test-name</td>
<td>Name of a group of unit tests specified in the control file.</td>
</tr>
<tr>
<td>codeCoverage</td>
<td>Specifies whether code coverage is enabled or disabled in the JadeUnitTestBatch application. This argument is optional, as you can also specify whether to enable code coverage in the control file.</td>
</tr>
<tr>
<td>profile</td>
<td>Specifies whether unit test profiling is enabled or disabled in the JadeUnitTestBatch application. This argument is optional, as you can also specify whether to enable unit test profiling in the control file.</td>
</tr>
<tr>
<td>log-file-name</td>
<td>Name of a log file used for error and test output. This argument is optional, as you can also specify the output destination in the control file.</td>
</tr>
</tbody>
</table>

The control file has the following format.

```
<unitTest>
   <reporter schema="schema name" class="class name" logFile="file name"/>
   <schema name="test schema name"/>
   <options deltaName="delta name" logCallStack="true" codeCoverage="true" profile="true"/>
   <tests>
      <test name="test name">
         ...
      </test>
   </tests>
</unitTest>
```

The file can also have the following standard XML declaration header.

```
<?xml version="1.0" encoding="utf-8"?>
```
The `<reporter>` XML element is optional, and is used to specify where the unit test output goes. If a `schema` and `class` name are specified, they must correspond to a class that implements the `JadeTestListenerIF` interface. A transient instance of this class is created and the unit test results are passed to the respective interface methods as the tests are run.

If a `logFile` is specified and is valid, the unit test results are written to this file. If both the `schema` and `class` name are specified, they take precedence and the `logFile` value is ignored. If neither the `schema` nor `class` name are specified or are invalid, error output and test result output are written to the log file specified as the third argument in the `Jadclient` command. If the log file is not specified or is invalid, output is written to a file named `JadeUnitTest.log` in the directory specified by the `DefaultLogDirectory` parameter in the [JadeMonitorBackground] section of the JADE initialization file.

The `<schema>` XML element is optional, and is used to specify the default schema in which the unit test classes are defined.

The `<options>` XML element is optional. You can use it to specify multiple options; for example, `<options deltaName="delta name" logCallStack="true" codeCoverage="true"/>`.

- The `deltaName` option indicates that unit tests and methods checked out to the specified delta should be run.
- The `logCallStack` option indicates whether the call stack is logged when an assert or exception occurs. By default, the call stack is logged; that is, `logCallStack="true"`.
- The `codeCoverage` and `profile` options indicate whether the unit tests have code coverage and profiling, respectively. By default, code coverage and profiling are not enabled; that is, `codeCoverage="false"` and `profiling="false"`, respectively. (You can also specify on the command line that code coverage and profiling are enabled.)

The `<tests>` XML element, which must be present, contains one or more `<test>` XML elements. Each `<test>` XML element is specified, identifying at least one unit test class to be run.

Tests can also include the contents of other tests, by using the `<includeTest>` element, as shown in the following example.

```xml
<includeTest name="test name"/>
```

This copies the contents of the test specified by the `name` attribute into the current test.

The following example shows the structure of a typical `<test>` XML element.

```xml
<unitTest>
  <options deltaName="dawn" logCallStack="true"/>
  <tests>
    <test name="DailyTests">
      <schema name="UnitTestTestSchema"/>
      <class name="UnitTests" includeSubclasses="true"/>
      <class name="WithdrawalTests">
        <method name="overdrawn"/>
        <method name="exceedsLimit"/>
      </class>
    </test>
  </tests>
</unitTest>
```

The `<schema>` XML element, which must have a `name` attribute corresponding to a valid schema, specifies the unit tests to be run for classes in the schema. The `<schema>` XML element has the optional attributes shown in the following examples.

```xml
<schema name="DemoUnitTestSchema" runAllTests="true"/>
<schema name="DemoUnitTestSchema" runAllTests="true" includeSubschemas="true"/>
```
The default value of the `runAllTests` attribute is `false`. If the `runAllTests` attribute value is `true`, all unit test methods of all subclasses of the `JadeTestCase` class in this schema are run as part of the test. In this case, any `<class>` XML elements in the test are ignored. The default value of the `includeSubschemas` attribute is `false`. If the `includeSubschemas` attribute value is `true`, all unit test methods of all subclasses of the `JadeTestCase` class in this schema and its subclasses are run as part of the test.

The `<class>` XML element, which must have a `name` attribute corresponding to a valid subclass of the `JadeTestCase` class, specifies the unit tests to be run for the class. The variants of this class element are as follows.

- Class name only.

  ```xml
  <class name="test class1 name"/>
  ```

  In this case, all unit test methods of the class are run as part of the test.

- Class name with `<method>` children.

  ```xml
  <class name="test class2 name">
    <method name="test method1 name"/>
    <method name="test method2 name"/>
  </class>
  ```

  Each `<method>` XML element must have a `name` attribute corresponding to a valid unit test method within the class. When `<method>` child elements are present, only the specified methods are run in the test.

- Class name with a `name` attribute and an `includeSubclasses` attribute.

  ```xml
  <class name="test class3 name" includeSubclasses="true"/>
  ```

  The default value of the `includeSubclasses` option is `false`. If the value is `true`, all unit test methods of the class and all of its subclasses are run as part of the test. If the option `includeClasses` attribute is set to `true`, any `<method>` children for the `<class>` element are ignored.

The `jadclient` non-GUI client application returns zero (0) if there were no errors; otherwise it returns an exception code. For example, if one or more tests fail, the exit code is 1467, in which case you can check the `JadeUnitTest.log` file for details of the failed test or test. Error 1466 is an indication that an incorrect argument was specified; for example, an invalid schema name or test name, and so on.

### Running Batch Mode Unit Tests that Require GUI Components

The `JadeUnitTestGuiNoForms` application enables you to run unit tests that require GUI components (that is, does not display forms) in batch mode, using the JADE executable (`jade.exe`) with the same arguments as the `JadeUnitTestBatch` application. For more details, see "Running Unit Tests in Batch Mode", earlier in this chapter.

To run unit tests in batch mode as a GUI No Forms client application using `jade.exe`, specify the following arguments in the command line.

```
jade path=database-path
   schema=RootSchema
ini=jade-initialization-file
app=JadeUnitTestGuiNoForms
startAppParameters
control-file-name
   test-name
   [codeCoverage=true|false]
   [profile=true|false]
   [log-file-name]
endAppParameters
```
The following example specifies the JADE executable (jade.exe) command line for the GUI No Forms JadeUnitTestGuiNoForms application.

```
jade path=c:\jade\system schema=TestUnitTestSchema ini=c:\jade\system\jade.ini app=JadeUnitTestGuiNoForms appServer=MyAppServer appServerPort=1234 startAppParameters c:\jade\unittest.xml DailyTests codecov=true profile=true c:\jade\logs\unittest.log endAppParameters
```

### Running Unit Tests from an Application

You can run tests from code, by creating an instance of the **JadeTestRunner** class and using the `runTests` method, which executes the test methods of all test classes in the collection specified in the `tests` parameter. The `tests` parameter is a collection of unit test classes and individual unit test methods. For a unit test class included in the collection, all unit test methods are executed in turn. You can specify classes and method together.

The following example shows tests being run for the **TestCalculator** class.

```
vars
tests : ObjectArray;
jtr : JadeTestRunner;
begin
create tests transient;
tests.add(TestCalculator);
create jtr transient;
jtr.runTests(tests);
epilog
delete tests;
delete jtr;
end;
```

The following example shows tests being run for the **TestConvertor** class and for two unit test methods of the **TestCalculator** class.

```
vars
tests : ObjectArray;
jtr : JadeTestRunner;
begin
create tests transient;
tests.add(TestConvertor);
tests.add(TestCalculator::add);
tests.add(TestCalculator::divide);
create jtr transient;
jtr.runTests(tests);
epilog
delete tests;
delete jtr;
end;
```

The destination and format of the output from the `runTests` method depends on whether a test listener has been set up by using the `setTestListener` method of the **JadeTestRunner** class. A test listener is an object that implements the **JadeTestListenerIF** interface.

If a test listener is present, the implementation of the interface methods determines the destination and format of the output. If there is no test listener, test results are written to the Jade Interpreter Output Viewer window. The following output is produced from the **TestCalculator** unit tests without a test listener.

```
hello
FAIL: TestCalc::divide - assertEquals - expected 5 but actual= 4
```
Listening for Unit Test Results

Before you run unit tests, you can create and register a unit test listener, which is an object that implements the `JadeTestListenerIF` interface. The interface methods for this object are invoked at certain stages during the test run by the `setTestListener` method of the `JadeTestRunner` class. Relevant test information is passed as parameters to the interface methods of the test listener.

The purpose of a test listener is to determine what is done with the unit test results. For example, you can display them on a form or capture them to a file. There can be at most one test listener. If there is no test listener, test results are written to the Jade Interpreter Output Viewer window.

To present the unit test results on a form, the form must implement the `JadeTestListenerIF` interface. For details about implementing the `JadeTestListenerIF` interface for a class selected in the Class Browser of a user schema, see "Implementing an Interface" and "Adding and Maintaining Interfaces", in Chapter 14 of the JADE Development Environment User's Guide.

When the form loads, create an instance of the `JadeTestRunner` class to run the unit tests. Before the tests run, use the `setTestListener` method of the `JadeTestRunner` instance to register the form as the test listener. The following example shows how this is achieved in the `load` method.

```java
load() updating;
vars
tests : ObjectArray;
jtr : JadeTestRunner;
begin
    create tests transient;
tests.add(TestCalculator);
    create jtr transient;
jtr.setTestListener(self);
jtr.runTests(tests);
epilog
    delete tests;
delete jtr;
end;
```

To capture the results of a unit test run to a file, add a class in your user schema with a reference of type `File` and make the class implement the `JadeTestListenerIF` interface. Create an instance of the `JadeTestRunner` class to run the unit tests. Before the tests begin, create an instance of your class and use the `setTestListener` method of the `JadeTestListener` instance, to register the instance as the test listener, as shown in the following example.

```java
vars
tests : ObjectArray;
// Listener class implements JadeTestListener and has reference of type File
listener : Listener;
jtr : JadeTestRunner;
begin
    create listener transient;
    create listener.file transient;
    listener.file.mode := File.Mode_Append;
```
listener.file.fileName := "C:\UnitTests\results.txt";
create tests transient;
tests.add(TestCalculator);
create jtr transient;
jtr.setTestListener(listener);
jtr.runTests(tests);
epilog
    delete tests;
    delete listener.file;
    delete listener;
    delete jtr;
end;

The following methods are reimplemented by a listener object.

- **The start** method is invoked when a unit test run begins.

  The number of unit test methods to be executed is determined and passed to the method, and can be used to display progress. For example, if there are 11 unit test methods to be executed and the test listener is a form containing a progress bar, the progress bar could be initialized showing that zero (0) out of 11 tests have been carried out.

  start(numberOfTestMethods: Integer);

- **The methodSuccess** method is invoked if all assertions in a unit test method succeed. It is passed the name of the successful test method.

  The method name parameter contains the fully qualified name of the unit test method in the form schema-name::class-name::method-name.

  methodSuccess(testMethodName: String);

The events during a unit test run that invoke callback methods are shown in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>When the listener method is invoked …</th>
</tr>
</thead>
<tbody>
<tr>
<td>finish</td>
<td>After the last test method for the last JadeTestCase subclass completes</td>
</tr>
<tr>
<td>message</td>
<td>Before the first test method for a JadeTestCase subclass starts or after the last test method completes</td>
</tr>
<tr>
<td>methodSuccess</td>
<td>If a test method completes successfully without an exception being raised or an assertion failing</td>
</tr>
<tr>
<td>start</td>
<td>Before the first test method for the first JadeTestCase subclass starts</td>
</tr>
<tr>
<td>testFailure</td>
<td>If a test method results in an exception being raised or an assertion failing</td>
</tr>
<tr>
<td>testSkipped</td>
<td>If a test method has the unitTestIgnore method option and is skipped</td>
</tr>
<tr>
<td>testSuccess</td>
<td>For each individual assertion that passes, in each test method run</td>
</tr>
</tbody>
</table>
Example of Using the JADE Unit Test Framework

The following diagram shows the classes used in the Calculator example of the use of the JADE unit testing framework.

The main classes in the Calculator example are:

- **Calculator**, which is a class with calculator-style functionality
- **TestCalculator**, which is the unit test class for testing Calculator functionality
- **Listener**, which is a listener class that outputs unit test results to a file on disk

The Calculator class, which has an attribute of type Integer, provides the following methods.

```pascal
add(n: Integer) updating;
begin
    result := result + n;
end;

clear() updating;
begin
    // Cleans the result
    result := 0;
end;

divide(n: Integer) updating;
begin
    ...
end;
```
result := result div n;
end;

getResult(): Integer;
begin
  return result;
end;

multiply(n: Integer) updating;
begin
  // not implemented yet
end;

square() updating;
begin
  result := result * result;
end;

squareRoot() updating;
begin
  //Bug: loops indefinitely
  while true do
  endwhile;
end;

subtract(n: Integer) updating;
begin
  //Bug: should be result = result - n
  result := result - 1;
end;

switchOff();
begin
  write "bye bye";
  // Display "bye bye", beep, switch off the screen
end;

switchOn() updating;
begin
  // Switch on the screen, display "hello", beep
  // and do other things that calculators do nowadays
  write "hello";
  result := 0;
end;

The TestCalculator class, which is a subclass of the JadeTestCase class and which has a reference called calculator of type Calculator, provides the following methods.

add() unitTest;
begin
  calculator.add(1);
  calculator.add(1);
  assertEquals(calculator.getResult(), 2);
end;

clearBeforeTest() unitTestBefore;
begin
calculator.clear;
end;

divide() unitTest;
begin
   // BUG 8 div 2 is 4 not 5
   calculator.add(8);
calculator.divide(2);
   assertEquals(calculator.getResult(), 5);
end;

divideByZero() unitTest;
begin
   expectedException(4030);
calculator.divide(0);
end;
multiply() unitTestIgnore; // not ready yet
begin
   calculator.add(10);
calculator.multiply(10);
   assertEquals(calculator.getResult(), 100);
end;

square() unitTest;
begin
   calculator.add(10);
calculator.square();
   assertEquals(calculator.getResult(), 100);
end;

subtract() unitTest;
begin
   calculator.add(10);
calculator.subtract(2);
   assertEquals(calculator.getResult(), 8);
end;

turnOnCalculator() unitTestBeforeClass, updating;
begin
   create calculator transient;
calculator.switchOn;
end;

turnoffCalculator() unitTestAfterClass, updating;
begin
   calculator.switchOff;
delete calculator;
end;

The Listener class, which implements the JadeTestListenerIF interface in the user schema and which has a reference of type File, provides the following implementation of the interface methods.

finish(elapsedTime: Time; testsFailed: Integer; testsSkipped: Integer;
       testsSucceeded: Integer);
begin
    file.writeLine(Tab & "Elapsed time=" & elapsedTime.String &
                    " failed=" & testsFailed.String &
                    " skipped=" & testsSkipped.String &
                    " succeeded=" & testsSucceeded.String);
end;

message(messageText: String);
begin
    // not implemented
end;

start(numberOfTestMethods: Integer);
vars
ts: TimeStamp;
begin
    file.writeLine("UNIT TEST RUN " & ts.String &
                    " (# " & numberOfTestMethods.String & " tests)");
end;

testFailure(testMethodName: String; callStack: String; failureReason: String);
begin
    file.writeLine(Tab & "Fail" & Tab & testMethodName &
                    " (" & failureReason & ")");
    file.writeLine(callStack);
end;

testSkipped(testMethodName: String);
begin
    file.writeLine(Tab & "Skip" & Tab & testMethodName);
end;

testSuccess(testMethodName: String);
begin
    file.writeLine(Tab & "Success" & Tab & testMethodName);
end;

The following JadeScript method runs the calculator unit tests and appends the results to a file C:\UnitTests\results.txt.

captureToFile();
vars
tests : ObjectArray;
listener : Listener;
jtr : JadeTestRunner;
begin
    create listener transient;
    create listener.file transient;
    listener.file.mode := File.Mode_Append;
    listener.file.fileName := "C:\UnitTests\results.txt";
    create tests transient;
    tests.add(TestCalculator);
    create jtr transient;
    jtr.setTestListener(listener);
    jtr.runTests(tests);
Code Coverage

Code coverage is a measure used in software testing to describe the degree to which the methods in a system have been tested. It is a useful measure to assure the quality of a set of tests, as opposed to directly reflecting the quality of the system under test.

Code coverage can help testers and developers to:

- Discover methods and blocks of code that are not exercised by a set of tests
- Create tests that increase code coverage
- Quantify the overall code coverage of a system, which is one measure of quality

When a test method is executed, some of the methods that have been written by the application developers are invoked.

A block of code represents an interpreter operation; for example, an assignment statement, method call, property access, if instruction, and so on. A block can contain nested blocks, so a single instruction can contain multiple blocks.

A method consists of instructions, which in turn consist of blocks of code. For example, each of the following assignments correspond to a single block of code.

```java
addr := "21 Somewhere Street";
addr := agent.addr1;
```

The following assignment requires a method to be executed before the assignment can be made. It corresponds to two blocks of code.

```java
addr := agent.getAddress1();
```

All of the blocks of code may not be executed in a single execution of the method if it contains:

- An if instruction with a condition that is not met
- A return instruction that exits early from the method

In such cases, the testing could be expanded to call the method a number of times so that all pathways through the method are executed and therefore all blocks are executed.

A measure of the code coverage is obtained by comparing the number of blocks in a method that have been executed with the total number of blocks for all instructions in the method.

Collecting Code Coverage Results

When you are running an application within the JADE development environment, you can use menu options from the Jade User Interrupt Code Coverage submenu to dynamically collect and report on code coverage results. For details, see "Determining Code Coverage", in Chapter 1 of the JADE Runtime Application Guide.
When determining code coverage by using the Jade User Interrupt submenu, the application must already be running. Only those methods that are executed after code coverage has started are reported on.

You can also code instructions within your application that use methods provided in the JadeProfiler class to start and stop collecting code coverage results and to report on the statistics.

You can also enable or disable code coverage from the File menu on the Unit Test Runner form. For details, see "Enabling Code Coverage" under "Performing File Menu Command Actions", earlier in this chapter.

Whilst you can gather code coverage results for any application, the main use is in measuring the quality of a suite of software testing methods.

The code coverage methods in the JadeProfiler class are used as follows.

- The startCodeCoverage method is called in the setup method before the testing starts.
- The stopCodeCoverage method is called in the teardown method after the testing ends.
- The reportCodeCoverage method is called to output the code coverage results to a file. If there is an existing file, records are appended to the file.
- The viewCodeCoverage method is called to stop the code coverage session, automatically initiate the code coverage application, and display the created code coverage result file in the Code Coverage Results Browser.

For details, see Volume 1 of the JADE Encyclopaedia of Classes.

**Viewing Code Coverage Results**

To display code coverage results:

- Select the Code Coverage command from the File menu in the JADE development environment.
- Select the View command from the Jade User Interrupt Code Coverage submenu in a running JADE application.
- Call the JadeProfiler class viewCodeCoverage method.

For details, see:

- "Viewing Code Coverage Results", in the JADE Runtime Application Guide
- "Viewing Code Coverage Results", in the JADE Development Environment User's Guide
- "Code Coverage Results Browser", in the following section
Code Coverage Results Browser

The Code Coverage Results Browser enables one or more coverage output files to be loaded and the code coverage results to be displayed, as shown in the following image.

The **Entity** list box in the top-left panel displays the schema entities for which coverage information has been recorded in the file or files that are loaded. Hierarchical and flat (that is, schema, class or primitive type, and method) views are available by using the commands in the View menu.

If you specify text in the **Entity** text box above the entity list, the first entry in the list that contains that string is selected (the search is not case-sensitive). Use the **Find Next** button to find the next entry in the list that contains that text. If the specified string cannot be found, a message is displayed in the status bar at the bottom of the browser.

The table at the top right of the browser displays code coverage information for the entity selected in the **Entity** list box.

In the hierarchical view, each parent level displays consolidated coverage information for all of its child levels (for example, the top coverage entity displays overall coverage information, each schema entity displays coverage information for all of its child types (that is, classes and primitive types), and each type entity displays coverage information for all of its child methods). In this view, entries and levels are added to and removed from the list box dynamically as you expand and collapse levels in the tree. This enables the list box to handle files with a large number of items; potentially more items than can be displayed in a list box.
In the flat views, each element displays coverage information for itself only (that is, there is no roll-up of information). As flat views load all elements into the list box and table, the flat method view can display a maximum of 32,000 items. If any view exceeds the maximum number of entries, a message box is displayed and the view displays up to the maximum number of entries only.

The columns in the top-right table are listed in the following table.

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>% of JADE Methods Executed</td>
<td>Percentage of the number of JADE methods defined on the type (class or primitive type) or schema that were executed. The entry is empty if the selected entity list item is not a type or schema.</td>
</tr>
<tr>
<td></td>
<td>You must manually request the percentage value for a schema must be manually requested (by selecting the <strong>Calculate Total Methods Executed Percent for Schema</strong> command from the File menu).</td>
</tr>
<tr>
<td>Total Blocks in executed methods</td>
<td>Total number of blocks that are available for execution by the executed methods for the selected entity. (Each method consists of a number of executable blocks.)</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> For schema, class, and primitive type entries, this does not include the number of blocks for methods that are not yet executed.</td>
</tr>
<tr>
<td>Covered Blocks</td>
<td>Number of blocks that have been executed at least once. If a single block is executed multiple times, it is still counted as a single covered block.</td>
</tr>
<tr>
<td>Covered Blocks %</td>
<td>Number of covered blocks as a percentage of the total blocks. Note that for schema, class, and primitive type entries, this does not include the number of blocks for methods that are not yet executed.</td>
</tr>
<tr>
<td>Not Covered Blocks</td>
<td>Number of blocks not executed at all (that is, total blocks minus the number of covered blocks)</td>
</tr>
<tr>
<td>Not Covered Blocks %</td>
<td>Number of blocks not executed as a percentage of the total blocks.</td>
</tr>
<tr>
<td>Executed Count</td>
<td>Total number of times a method was executed.</td>
</tr>
</tbody>
</table>
The information displayed in the pane at the bottom of the Code Coverage Results Browser depends on the selected element in the Entity list box at the top left of the browser.

- When the selected element is the top-level **Coverage** item, the pane displays nothing.
- When the selected element is a schema, the pane displays overall class and primitive type coverage information for the selected schema, as shown in the following image.

![Code Coverage Results](image)

**Note** The figures for the total number of JADE methods in the schema, the methods included in the coverage, and the methods not included in the coverage are displayed only after you select the **Calculate Total Methods Executed Percent for Schema** command from the File menu for that schema entity.

- When the selected element is a class or primitive type, the pane displays overall coverage information for the class or primitive type, including the JADE methods (but excluding any external methods) that do not appear in the code coverage results.
In the following image of an example of code coverage results for a class element, three JADE methods defined in the FormClientApp class do not appear in the code coverage results (that is, they have not been executed at all).
When the selected element is a method, the pane displays the method source with a light-red background color, indicating the lines of code that have been covered, as shown in the following image.

A line is highlighted if it contains at least one block that has been executed. There is no indication of partially executed lines; that is, where a line contains multiple blocks but only some of the blocks have been executed. A line of code is highlighted entirely or not at all.

The Code Coverage Results Browser can load coverage output files that have been captured in a different environment from the one in which the Code Coverage Results Browser is running. The browser can display information about schemas, types, and methods that it cannot find because they do not exist in that environment. It can also display entities that are different from the entities for which the output was recorded.

When a file is loaded, the following checks are performed.

- The browser attempts to find an entity using its fully qualified name; that is, it attempts to find methods using the `schema-name::type-name::method-name` format.

- An entity that does not exist in the environment of the Code Coverage Results Browser is displayed in the Entity list box with a red cross icon.

Coverage information for such items is still displayed in the table at the top right of the browser (because it comes entirely from the coverage output file), but no additional information can be displayed in the lower pane. (An appropriate status line message indicates this.) For example, if the Movie schema does not exist in the browser’s environment at all, all of its child items are marked with a cross and the lower pane does not show any source code coverage for the selected `displayExpandedTitle` method.

- If an entity is found using the fully qualified name, the object identifier and edition of the entity are compared.
against the object identifier and edition recorded in the code coverage output file (that is, the entity in the environment of the browser is checked against the object identifier and edition of the entity for which the coverage information was recorded). If they match, everything is in order so all code coverage information for this entity can be displayed.

If the object identifier or the edition do not match, the entity is different from the one for which coverage results were recorded. For example, a method may have been changed and recompiled since the coverage information was captured or coverage information obtained in an environment running a later version of a schema could be loaded into a Code Coverage Results Browser running from an environment containing an older version of the schema.

In these cases, entities are displayed in the Entity list box with an exclamation mark icon. All coverage information is still displayed, including the source code information in the lower pane. However, the source code highlighting may be invalid if the method has changed significantly since the code coverage information was recorded. (An appropriate status line message is displayed in these situations.)

**Using Code Coverage Menus**

The Code Coverage Results Browser provides the menus listed in the following table.

<table>
<thead>
<tr>
<th>Menu</th>
<th>Provides commands that enable you to …</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>Perform file-related tasks and to exit from the browser</td>
</tr>
<tr>
<td>View</td>
<td>Customize the display of code coverage results</td>
</tr>
</tbody>
</table>

For details, see the following subsections.

**Performing File Menu Command Actions**

The File menu provides the commands listed in the following table.

<table>
<thead>
<tr>
<th>Command</th>
<th>For details, see…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>Loading Code Coverage Files</td>
</tr>
<tr>
<td>Clear Session</td>
<td>Clearing Session Information</td>
</tr>
<tr>
<td>Save As CSV</td>
<td>Saving Code Coverage Results in CSV Format</td>
</tr>
<tr>
<td>Save Schema Methods Not in Coverage as CSV</td>
<td>Saving Methods not in the Code Coverage</td>
</tr>
<tr>
<td>Calculate Total Methods Executed Percent for Schema</td>
<td>Calculating the Total Methods Executed as a Percentage</td>
</tr>
<tr>
<td>Exit</td>
<td>Exiting From the Code Coverage Results Browser</td>
</tr>
</tbody>
</table>

**Loading Code Coverage Files**

To load a code coverage file into the Code Coverage Results Browser

Select the Load command from the File menu.

The Load submenu is then displayed. For details, see the following subsections.
Loading and Merging Coverage for Methods

1. Select the Load command from the File menu.

2. Select the Load and Merge Coverage for Methods command from the Load submenu that is then displayed.

3. In the Open common dialog that is then displayed, specify the name in the File name combo box of the coverage file (that is, a .ccd file) that you want to load from the directory selected in the Look in combo box and then click the Open button. (Alternatively, click the Cancel button to abandon your selection.)

During the load operation, progress information is displayed in the status bar. You can abort the load process by clicking the Cancel button that is displayed at the right of the status bar.

The information in the specified file is merged with the currently loaded information in the Code Coverage Results Browser (assuming that there is some). If no information is currently loaded, the file is loaded to create a new session.

When merging a file (that is, loading a file into a non-empty session), coverage information may be loaded for methods for which there is already coverage information (in the current session). In such cases, method editions are used to determine which has the latest coverage information; that is, the current session or the file being loaded.

If the current session already has coverage information for a method and the method editions are the same, the new coverage information for the method is merged with the existing information.

Loading and Replacing Coverage for Methods

To load a code coverage file and replace existing coverage

1. Select the Load command from the File menu.

2. Select the Load and Replace Coverage for Methods command from the Load submenu that is then displayed.

3. In the Open common dialog that is then displayed, specify the name in the File name combo box of the coverage file (that is, a .ccd file) that you want to load from the directory selected in the Look in combo box and then click the Open button. (Alternatively, click the Cancel button to abandon your selection.)

During the load operation, progress information is displayed in the status bar. You can abort the load process by clicking the Cancel button that is displayed at the right of the status bar.

If no information is currently loaded, the file is loaded to create a new session.

When loading a file into a non-empty session, coverage information may be loaded for methods for which there is already coverage information (in the current session). In such cases, method editions are used to determine which has the latest coverage information; that is, the current session or the file being loaded.

If the method edition in the currently loaded information is greater than the method edition in the file, the assumption is that the current session already has the latest coverage information and the data in the file (for this method only) is ignored.

If the method edition in the file is greater than or equal to the edition of the method in the current session, the assumption is that the file contains later information, so the current session information (for this method only) is discarded and recreated from the file being loaded.
Clearing Session Information

To clear session information from the Code Coverage Results Browser

- Select the Clear Session command from the File menu.

The current session information is cleared so that a new code coverage output file can be loaded from scratch.

Saving Code Coverage Results in CSV Format

To save code coverage results to a Comma-Separated Values (CSV) file

- Select the Save As CSV command from the File menu.

The Save As common dialog is then displayed, prompting you to specify a file name. The current session information is then saved to the specified file in CSV format.

Regardless of the selected view, the information is written to the CSV file in hierarchy order, with the following columns.

- Schema
- Type (that is, a class or a primitive type)
- Method
- Total Blocks
- Covered Blocks
- Covered Blocks %
- Not Covered Blocks
- Not Covered Blocks %
- Covered Seconds

Saving Methods not in the Code Coverage

To save methods not in the code coverage

- Select the Save Schema Methods Not in Coverage as CSV command from the File menu.

  This command is enabled only when a schema is selected in the Entity list box.

The Save As common dialog is then displayed, prompting you for a file name and the file path, if you want it saved to a location other than the default bin directory. A list of all JADE methods in this schema that have not been executed is then saved to a CSV file in a table containing the Schema, Type, and Method columns. The type column can contain a class or a primitive type entity. The values that are output to this file are cleared when you perform another file load operation.

After the file has been produced, the total number of JADE methods in the schema, the number of methods included in the coverage, the number of methods not included in the coverage, and the total percentage of JADE methods executed is displayed at the left of the pane at the bottom of the Code Coverage Results Browser.
Calculating as a Percentage the Total Methods Executed

To calculate as a percentage the total methods executed

- Select the Calculate Total Methods Executed Percent for Schema command in the File menu. This command is enabled only when a schema is selected in the Entity list.

The total percentage of JADE methods executed for this schema is then calculated and displayed in the pane at the bottom of the Code Coverage Results Browser. In addition, the total number of JADE methods in the schema, the number of methods included in the coverage, and the number of methods not included in the coverage is included in the display.

The values that are displayed are cleared when you perform another file load operation.

**Note** This operation is manual because the calculation could take a significant amount of time for a large schema.

Exiting From the Code Coverage Results Browser

- Select the Exit command from the File menu.

The Code Coverage Results Browser is then closed.

Performing View Menu Command Actions

The View menu provides the commands listed in the following table.

<table>
<thead>
<tr>
<th>Command</th>
<th>For details, see…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchy</td>
<td>Viewing a Schema, Type, Method Hierarchy</td>
</tr>
<tr>
<td>Schemas Only</td>
<td>Viewing Schemas Only</td>
</tr>
<tr>
<td>Types Only</td>
<td>Viewing Types Only</td>
</tr>
<tr>
<td>Methods Only</td>
<td>Viewing Methods Only</td>
</tr>
<tr>
<td>Sort By Name</td>
<td>Sorting By Name</td>
</tr>
<tr>
<td>Sort By Total Blocks</td>
<td>Sorting By Total Blocks</td>
</tr>
<tr>
<td>Sort By Covered Blocks</td>
<td>Sorting By Covered Blocks</td>
</tr>
<tr>
<td>Sort By Covered Blocks %</td>
<td>Sorting By Covered Blocks Percentage</td>
</tr>
<tr>
<td>Ascending</td>
<td>Sorting in Ascending Order</td>
</tr>
<tr>
<td>Descending</td>
<td>Sorting in Descending Order</td>
</tr>
<tr>
<td>Display List of Entities Not Covered</td>
<td>Displaying a List of Entities that are not Covered</td>
</tr>
</tbody>
</table>

Viewing a Schema, Type, Method Hierarchy

To view a schema, type, method code coverage hierarchy

- Select the Hierarchy command from the View menu.

The default hierarchy view of code coverage is then displayed; that is, entries are added and removed in the Entity list box dynamically, as you expand and collapses levels in the hierarchy tree.
Each parent level displays consolidated coverage information for all of its child levels; that is, the top Coverage element displays overall coverage information, a schema element displays coverage information for its child classes and primitive types, and a type element displays coverage information for its child methods.

**Viewing Schemas Only**

- To view schema items only
  - Select the Schema Only command from the View menu.

  The Code Coverage Results Browser then displays a flat view of schema items only. You cannot expand the schema element to view code coverage information at a lower level.

**Viewing Types Only**

- To view class and primitive type items only
  - Select the Types Only command from the View menu.

  The Code Coverage Results Browser then displays a flat view of type items only. You cannot expand the schema element to view code coverage information at a lower level.

**Viewing Methods Only**

- To view method items only
  - Select the Methods Only command from the View menu.

  The Code Coverage Results Browser then displays a flat view of method items only.

**Sorting By Name**

- To sort a flat view of items by name
  - Select the Sort By Name command from the View menu. This command is enabled only when a schema, type, or method flat view of code coverage results is displayed.

  The list of elements is then sorted by name in ascending or descending order, depending on whether the Ascending or Descending option is selected in the View menu.

**Sorting By Total Blocks**

- To sort a flat view of items by total blocks
  - Select the Sort By Total Blocks command from the View menu. This command is enabled only when a schema, type, or method flat view of code coverage results is displayed.

  The list of elements is then sorted by total blocks in ascending or descending order, depending on whether the Ascending or Descending option is selected in the View menu. For details about total blocks, see "Code Coverage Results Browser", earlier in this chapter.
Sorting By Covered Blocks

To sort a flat view of items by covered blocks

- Select the Sort By Covered Blocks command from the View menu. This command is enabled only when a schema, type, or method flat view of code coverage results is displayed.

The list of elements is then sorted by covered blocks in ascending or descending order, depending on whether the Ascending or Descending option is selected in the View menu. For details about covered blocks, see "Code Coverage Results Browser", earlier in this chapter.

Sorting By Covered Blocks %

To sort a flat view of items by covered blocks as a percentage of the total blocks

- Select the Sort By Covered Blocks % command from the View menu.

This command is enabled only when a schema, type, or method flat view of code coverage results is displayed.

The list of elements is then sorted by covered blocks as a percentage of the total blocks in ascending or descending order, depending on whether the Ascending or Descending option is selected in the View menu.

For details about total blocks, see "Code Coverage Results Browser", earlier in this chapter.

Sorting in Ascending Order

To sort a flat view of items in ascending order

- Select the Ascending command from the View menu. This command is enabled only when a schema, type, class, or method flat view of code coverage results is displayed.

The list of elements is then sorted in ascending order.

Sorting in Descending Order

To sort a flat view of items in descending order

- Select the Descending command from the View menu. This command is enabled only when a schema, class, or method flat view of code coverage results is displayed.

The list of elements is then sorted in descending order.

Displaying a List of Entities that are not Covered

To display a list of entities that are not covered

- Select the Display List of Entities Not Covered command from the View menu.

The display of entities that are not covered in the pane at the bottom of the Code Coverage Results Browser is then toggled. When you select this command from a:

- Schema entity and the option is enabled, the list of types with JADE methods that have not been included in the coverage is displayed.

- Class or primitive entity and the option is enabled, the list of JADE methods that have not executed is displayed.
Chapter 18  Tracking Methods

This chapter contains the following topics.

- Overview
- Enabling Tracking
- Tracking a Method
- Helper Methods for Obtaining Method Tracking Information
- Examples of Using Method Tracking

Overview

JADE method tracking enables you to identify a target method to be tracked and to intercept the execution of that method by executing a *preamble* method before the target method is executed and by executing a *postamble* method after the target method has returned. There are no coding changes required in the target method for this to happen.

The following diagram shows the flow of execution with and without method tracking taking place.

![Diagram showing the flow of execution with and without method tracking](image)

From the *preamble* and *postamble* methods, you can determine the following information.

- Name of the tracked method
- Receiver object
- Parameters
- Return value

Enabling Tracking

The *MethodTrackingEnabled* parameter in the [JadeSecurity] JADE initialization file used by the database server controls whether method tracking is allowed. Set this parameter to *true* to enable method tracking for all processes on all nodes.
The value of the `MethodTrackingEnabled` parameter is checked for a process the first time the process calls the `startMethodTracking` method of the `Process` class.

**Tip** You can change the value of this parameter in the JADE initialization file at any time, to alter the tracking enabled setting without having to stop and restart the database server node.

## Tracking a Method

Use the `startMethodTracking` method of the `Process` class to initiate method tracking for the receiving process. If tracking is not enabled, an attempt to use the `startMethodTracking` method results in an exception being raised.

The receiving `Process` instance can be any current process, including the current process. The parameters of the `startMethodTracking` method are listed in the following table.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>targetMethod</code></td>
<td>Method to be tracked.</td>
</tr>
<tr>
<td><code>preambleMethod</code></td>
<td>Method to be invoked just before calling the target method.</td>
</tr>
<tr>
<td><code>postambleMethod</code></td>
<td>Method to be invoked just after returning from executing the target method (that is, after any epilog code in the target method has been executed).</td>
</tr>
<tr>
<td><code>receiver</code></td>
<td>Receiver for the preamble and postamble methods. The receiver object must be accessible by the target process.</td>
</tr>
</tbody>
</table>

The following methods cannot be tracked.

- `getAndValidateUser` in the `Global` class or a reimplementation in a subclass
- `isUserValid` in the `Global` class or a reimplementation in a subclass

Method tracking is not currently supported for `serverExecution` methods.

**Note** To avoid repeated calls and kernel stack overflow exceptions, the tracking method should not track itself or any of the methods that it calls.

The `stopMethodTracking` method of the `Process` class turns off method tracking of the target method by the receiving process.

## Helper Methods for Obtaining Method Tracking Information

The `preambleMethod` and `postambleMethod` methods must have the following signature:

```java
   method(paramList: ParamListType);
```

When invoked, the `paramList` parameter contains a list of parameters matching those of the method being tracked.
You can use the methods listed in the following table to obtain information about the method being tracked.

<table>
<thead>
<tr>
<th>Method</th>
<th>Information Obtained</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application::getParamListTypeEntry</td>
<td>Value of the parameter with the specified index within the paramList parameter</td>
</tr>
<tr>
<td>Application::getParamListTypeLength</td>
<td>Number of entries contained in the paramList parameter</td>
</tr>
<tr>
<td>Process::getTrackedMethod</td>
<td>Tracked method for which the preamble or postamble method has been invoked</td>
</tr>
<tr>
<td>Process::getTrackedMethodReceiver</td>
<td>Object used as the receiver for the method being tracked</td>
</tr>
<tr>
<td>Process::getTrackedMethodReturnValue</td>
<td>Return value of the method being tracked</td>
</tr>
<tr>
<td>Process::getTrackedMethodStatus</td>
<td>Value representing the current state of the tracked method; that is, whether it is about to be called, has just returned normally, or has just exited abnormally</td>
</tr>
</tbody>
</table>

Examples of Using Method Tracking

For method tracking to be enabled, the following entry is made in the JADE initialization file used by the database server.

```
[JadeSecurity]
MethodTrackingEnabled = true
```

The following example initiates method tracking for a method defined on the `Customer` class, where the `preamble` and `postamble` methods are defined on the `Process` class in the same schema as the tracked method.

```
startTracking(targetProcess : Process);
vars
begin
    targetProcess.startMethodTracking(Customer::getAddress, // tracked method
                                       Process::beforeCall, // preamble method
                                       Process::afterCall, // postamble method
                                       targetProcess); // receiver
end;
```

In the following example, method tracking is initiated for a method defined in a different schema from where the `preamble` and `postamble` methods are defined. The `Tracker` class is defined within the user schema and the `tracker` parameter is a persistent instance of the `Tracker` class.

```
initiateTracking(targetProcess: Process, tracker: Tracker);
vars
    scm : Schema;
    cls : Class;
    mth : Method;
begin
    scm := rootSchema.getSchema("AcctSchema");
    cls := scm.getClass("Customer");
    mth := mth.getMethod("getAddress");
    targetProcess.startMethodTracking(mth, // tracked method
                                       Tracker::beforeCall, // preamble method
                                       Tracker::afterCall, // postamble method
                                       tracker); // receiver
end;
```
Define the preamble and postamble methods in the Tracker class. The afterCall postamble method in the following example uses simple write instructions to provide tracking information.

```pascal
afterCall(paramList: ParamListType);
vars
  status : Integer;
  num, i : Integer;
begin
  // Status information - Is this the preamble or postamble?
  // Or has there been an exception?
  status := process.getTrackedMethodStatus;
  if status = 0 then write "no tracked method";
  elseif status = 1 then write "in preamble method";
  elseif status = 2 then write "in postamble method";
  elseif status = 3 then write "exception occurred";
  endif;
  // Tracked method
  write "tracked method: " & process.getTrackedMethod.name;
  // Receiver
  write "receiver: " & process.getTrackedMethodReceiver.String;
  // Parameters
  num := app.getParamListTypeLength(paramList);
  foreach i in 1 to num do
    write "parameter " & i.String & ": "
    & app.getParamListTypeEntry(i, paramList).String;
  endforeach;
  // Return value
  write "return value: " & process.getTrackedMethodReturnValue.String;
end;
```
Chapter 19  Tracing Transactions

This chapter contains the following topics.

- Overview
- Starting and Stopping Transaction Tracing
- Registering Properties for Transaction Tracing
- Analyzing Transaction Trace Information

Overview

Transaction tracing enables you to capture information about persistent objects that have been updated, created, or deleted within a transaction. You can also optionally register properties that will have information captured when they are updated.

Transaction tracing information is placed in a transient instance of the JadeTransactionTrace class associated with the current process. This instance is created, if it does not already exist, when the startTransactionTrace method of the Process class is called.

You can register callback methods to be invoked just before a transaction commits. These callback methods can report on and analyze the changes that have taken place within the current transaction.

**Note** You can clone or copy a JadeTransactionTrace object to a non-shared transient instance, by using the copySelf or cloneSelf methods, respectively. This enables you to retain tracing information from previous transactions.

When cloning or copying, the myProcess reference is not set up. To obtain this reference, use the getEntry and getEntryCount methods on the cloned or copied object.

Starting and Stopping Transaction Tracing

Transaction tracing is started and stopped by calls to the startTransactionTrace and stopTransactionTrace methods, respectively, for the current Process instance. The methods can be called regardless of the current transaction state.

If transaction tracing is started while a transaction is active, only objects updated, created, and deleted after tracing is initiated are recorded. Exceptions are raised if the startTransactionTrace method is called when transaction tracing has already been started or the stopTransactionTrace method is called when transaction tracing has not been started.

The following code fragments start and stop transaction tracing.

```
process.startTransactionTrace;

process.stopTransactionTrace;
```

Registering Properties for Transaction Tracing

The standard transaction tracing information consists of information about object creates, deletes, and updates.
You can optionally request that additional information is captured about updates to specific properties, using the `enableTransactionTracing` method of the `Property` class with the value `true` passed as the parameter, as shown in the following code fragment.

```cpp
Customer::status.enableTransactionTracing(true);
```

To unregister the property, call the `enableTransactionTracing` method passing the value `false`, as shown in the following code fragment.

```cpp
Customer::status.enableTransactionTracing(false);
```

### Analyzing Transaction Trace Information

You can register callback methods that are invoked just before a transaction is committed. To register a callback method, call the `enableTransTraceCallback` method of the `Process` class, passing the method to be invoked, the object used as the receiver, and a `Boolean` value of `true` to indicate registering.

```cpp
The following code fragment specifies that when a transaction for the current process commits, a method `Customer::commitCallback` is to be called for the receiver `cust`, which is of type `Customer`.
```

```cpp
process.enableTransTraceCallback(Customer::commitCallback, cust, true);
```

---

**Note** The callback method must have no parameters and no return type.

You can call the `enableTransTraceCallback` method multiple times, to register additional method callbacks when a transaction commits. Calling the `enableTransTraceCallback` with a method and receiver combination that has been previously registered is ignored.

Methods are invoked in reverse order of when they were registered; that is, the most recently registered methods are invoked first.

**Notes** The invoked method should not attempt to commit the transaction. Doing so causes repeated invocations of the method, leading eventually to a kernel stack overflow.

Similarly, the invoked method should not abort the current transaction. Doing so raises a `1026 (Not in transaction state)` exception after the method returns and an attempt to commit the transaction is made.

If an exception occurs within an invoked method and is not dealt with by an exception handler, the transaction is not committed.

The following example of a callback method uses functionality in the `JadeTransactionTrace` class to write information about the current transaction to the Jade Interpreter Output Viewer.

```cpp
commitCallback();
vars
    trace : JadeTransactionTrace;
    index : Integer;
    totalEntries : Integer;
    object : Object;
    operation : Integer;
    property : Property;
    value : Any;
begin
    trace := process.getTransactionTraceObject;
    totalEntries := trace.getEntryCount();
    write "-----------------------------------";
    write "Transaction ID: " & trace.tranId.String;
```
write "Started at: " & trace.startTime.String;
write "Total traced updates: " & totalEntries.String;
foreach index in 1 to totalEntries do
    write "",
    // retrieve the specific entry and write out the details
    trace.getEntry(index, object, operation, property, value);
    write "Traced Update: " & index.String;
    write "Object: " & object.String;
    write "Operation: " & operation.String & " - " &
        self.translateTraceOperation(operation);
    // if the entry is a property update, output that information as well
    if property <> null then
        write "Property: " & property.schemaType.name & "::"
            & property.name;
        write "Property Value: " & value.String;
    endif;
endforeach;
write "=================================================";
end;

The following output from the commitCallback example callback method is representative.

=================================================
Transaction ID: 5928
Started at: 18 February 2009, 14:15:40
Total traced updates: 2

Traced Update: 1
Object: Customer/2085.5
Operation: 3 - Object Update

Traced Update: 2
Object: Customer/2085.5
Operation: 7 - Manual Set Property
Property: Customer::status
Property Value: Bankrupt

=================================================
Chapter 20  Partitioning Database Files

This chapter contains the following topics.

- **Overview**
  - Partitioning Example
  - State of a Partition
- **Database Partitioned File Structures**
  - Partition Control File
  - Partition Index File
  - Partition Data Files
- **Partition Index, Modulus, and Creation Window**
- **Preparing to Partition a Database File**
  - Empty Database File
  - Non-Empty Database File
  - Flexibility in Partitioning Database Files
- **Partition Location**
- **Partitioning by Period**
- **Automatic Partitioning**
  - Partition Methods
- **Manual Partitioning**
- **Bulk Removal of Objects**
- **RootSchema Methods Related to Database Partitioning**
  - Class Methods
  - DbFile Property and Methods
  - JadeDbFilePartition Properties and Methods
  - Object Methods
  - System Method
- **Batch JADE Database Admin Utility Partitioning Commands**
Overview

Database file partitioning provides:

- Easier management of large amounts of historical data, particularly for bulk data operations such as backups and deletions
- Cost savings through the use of tiered disk storage, which enables you to split database files into partitions, with different partitions then being able to be stored on different media devices, tiered disk storage, or both different media devices and tiered disk storage
- The ability to move objects between partitions programmatically, which enables you to avoid accessing historical information if it is not required, thereby improving performance of online and batch operations

A logical database file can be split into several parts, or partitions, where each partition corresponds to a physical file at the file system level. A logical database file is identified by its schema-defined name and is addressable by its name or number. A file partition is addressable by a `DbFile` instance number and a unique partition identifier (`JadeDbFilePartition` class `partitionID` property), which is a positive `Integer64` value in the range 1 through 2^32-15. This unique identifier remains associated with a partition for its lifetime.

When the `setPartitioned` method is executed for an instance of the `DbFile` class representing a new database file, the required partition structures are established when the file is first instantiated. Making an existing database file partitioned requires a reorganization to convert the standard map file to the partitioned structure.

**Note** A prerequisite for partitioning a database file is that a single class is mapped to the file.

Partitioning a class enables you to locate instances in different partitions of a database file. A file partition is therefore a part of a database file containing a subset of instances of a single class.

When a database file has been partitioned, the set of partitions associated with a schema-defined map file is referred to as a **partition set**.

The external file name assigned to a file partition is derived from the schema-defined file name and incorporating the partition number as a qualifier, with the following format.

```
<map_file_name>_part<partition-number>.dat
```

For details about replicating partitioned file structures and most meta data on SDS secondary databases, see "SDS and Recovery Considerations when Using Partitioned Database Files", in Chapter 1 of the JADE Synchronized Database Service (SDS) Administration Guide.

The `JadeDbFilePartition` class provides methods that enable you to freeze or thaw a file partition. When a partition is frozen:

- The partition is converted to a read-only state, allowing its relocation to read-only media
- Objects read from the partition are marked as frozen, overriding individual object volatility state

When a partition is thawed:

- The read-only status is removed, allowing updates to objects residing in the partition
- Volatility reverts to transparent, removing the frozen override

**Partitioning Example**

A logical database file can be split into several component parts called partitions, where each partition corresponds to a physical file at the file system level.
In an unpartitioned database file, all instances of a specific class are wholly contained in a single database file, commonly referred to as a map file. A map file can host instances of more than one class. Partitioning a database file allows instances of the class to be located in different partitions of a database file. A file partition is therefore a part of a database file containing a subset of instances of a single class.

In the example in this chapter, there are two related classes, Order and OrderItem, which contain sales order data. These classes are mapped to their own dedicated map file. The database went live in January 2003 and sales order data has been captured and accumulated since that date.

The following diagram shows a partitioning scheme where Order and OrderItem objects are now spread across a number of file partitions where the actual partition is determined by a date attribute. In this image, the flat disks or platters represent file partitions. Each partition stores the Order and OrderItem objects for a specific monthly period.

Most data that tends to accumulate over time is a candidate for partitioning. A typical characteristic of this kind of data is that it tends to be comprised of objects that are created, read often, and not updated. Generic examples include application audit trails, transactions, messages, and events. Specific examples include sales or orders, banking transactions, test results, meter readings, and so on.
One of the benefits to partitioning large map files is that partitions containing historical objects that are never updated and are read infrequently can be moved to a lower-cost storage solution. Individual partitions can be frozen (made read-only) and can also be taken offline indefinitely or until required by the application.

A file reorganization mechanism can convert an existing non-partitioned database file into a partitioned format, enabling partitioning schemes like the one shown in the example to be applied retroactively.

**State of a Partition**

A file partition has two principal states that determine access to objects in the partition

- *Active*, in which objects can be created, updated, and deleted
- *Frozen* (or read-only), in which objects cannot be created, updated, and deleted

You can specify a different file-system location for each partition.

A frozen partition can be operationally marked as offline, which means that the file partition cannot be accessed through the file system. The distinction between an online and offline partition is significant when dealing with exception conditions. For example, a reference to an object in an offline partition is a normal condition whereas a reference to a supposedly online partition that is missing is an unexpected or abnormal exception.

System event trapping is one way to allow external management tools to take action to resolve the absent partition condition; the action may be as simple as asking an operator to mount a specified disk from an archive.

The state of a non-active partition overrides the state of objects contained in the partition. The frozen partition state overrides object volatility, which means that all objects contained in a frozen partition are themselves frozen. Similarly, all objects contained in an offline partition are considered to be offline (or not present).

If you attempt to update a frozen object, a 1106 (Cannot update a frozen object) exception is raised. If you attempt to reference an object, by using its object identifier (oid), located in an offline partition, a 12 (Cannot access object in an offline file or offline file partition) exception is raised. If you attempt access to an offline partition other than to read an object, a 3144 (Cannot access an offline file partition) exception is raised.
Database Partitioned File Structures

The following image shows the database control file and a partitioned database file.

In the database, a partitioned database file comprises the following physical files:

- A partition control file
- A partition index file
- Several partition data files

Partition Control File

The partition control file has an external name in the format `map–file–name.dat`. It is the control file for partitions in the same way as the database control file `_control.dat` is the control file for the entire database.

The partition control file contains a partition control record for each partition in the partition set. These records contain attributes such as the creation and last modified timestamps for the partition. The `partitionID` is used as the relative record offset to access the control record of a partition.

The file control record in `_control.dat` that corresponds to a partitioned file contains the attributes of the partition control file.

Use the batch JADE Database utility (`jdbutilb`) `restorePartitionFile` command to restore a backed up partitioned database file.
Partition Index File

The partition index file has an external name in the format `map-file-name.ndx.dat`. It is a global index used to look up the partition in which an object is located, using its object identifier (oid). Only the instance identifier and not the class identifier is stored, since a partitioned file contains instances of a single class and its subobjects.

Locating and centralizing the indexes in a separate file provides an efficient mechanism to locate an object in a partition using its object identifier (oid), and to determine whether an object resides in an offline partition.

When a class and database file is partitioned, subobjects are located in the same partition as their parent. Subobjects are exclusive collections, Unstructured Data Resources (UDRs), binary large objects (blobs), and string large objects (slobs). To determine the partition in which a subobject is located, the instance identifier of the parent is required. The parent instance identifier is directly available in the oid for exclusive collections, blobs, and slob and from the header oid attribute for collection blocks.

Partition Data Files

Partition files are the physical files used to store user objects.

A database file partition is addressable by a number (the value of the `number` property inherited from the `SchemaEntity` superclass of the `DbFile` class) and a unique partition identifier. A partition identifier is a positive integer in the range 1 through $2^{32}-15$ (4,294,967,296 minus 15). This unique identifier remains associated with a partition for the lifetime of the partition.

When a database file has been partitioned, the set of partitions associated with a schema-defined map file is referred to as a `partition set`. The external file name assigned to a file partition is derived from the schema-defined file name and incorporates the partition number as a qualifier, in the following format.

```
map-file-name_partpartition-number.dat
```

An instantiated partitioned file contains at least one partition. The first partition is assigned a partition identifier of one (1) and has the file name `map-file-name_part0000000001.dat`.

Further partitions are assigned numbers 2, 3, 4, and so on up to $2^{32}-15$ (15 instances are reserved). The latest partition created (that is, the partition with the highest assigned number) is the `current` (or default) partition.

Partition Index, Modulus, and Creation Window

When instances of a partitioned class are created, the file partition in which an instance is stored is determined by a zero-relative index referred to as the `partition index`.

A user-defined partition `modulus` (\(m\)) associated with a partitioned file defines a window of the \(m\) latest partitions in which new instances are stored. The latest partition of a modulus is referred to as the `creation window`, since it defines the window (a subset) of partitions where new objects are created. The partition modulus is an attribute of a partitioned file, with a value in the range 1 through 1,024. Each partitioned file can have a different value, to allow the partitioning schemes to vary.

The partition index value of zero (0) always refers to the latest partition created, partition 1 to the second-latest, and so on up to the limit of partition index \(m-1\). The latest partition is also referred to as the `current` partition. Whenever a new partition is added to the file, the `creation window` slides up by one, so that partition index zero (0) corresponds to the newly created partition and the `prior` partition with index `modulus - 1` drops out of the window.

The partition index concept allows partitioning algorithms to allocate new objects within a bounded subset of partitions. The algorithm can assume a fixed number of partitions and need not consider historical partitions.
The following diagram shows a partitioning scheme where the creation window has a modulus of five (5). New objects are stored in the five latest partitions and the valid range of partition index values is zero through four (0 - 4).

In this diagram, partition index zero (0) maps to the latest partition, which in this case is 100. If a new partition were to be created, the new partition would be assigned partition identifier 101 and partition index zero (0) would then correspond to partition identifier 101, as that is now the latest.

To implement a standard sliding window strategy in which a new partition is created periodically (for example, daily, weekly, or monthly), leave the modulus set to the default value of 1 and set the partition index to zero (0) for creates. This will cause objects to be created in the latest partition, by default. The partition with index value zero (0) is used unless you re-implement the autoPartitionIndex method or call the setPartitionIndex method.

Preparing to Partition a Database File

A database file can be partitioned only if:

- At most, one class is mapped to the file.

  Mapping more than one class to a partitioned file would defeat many of the benefits of partitioning, so this is explicitly disallowed. You can enable partitioning on a file before a singleton class is mapped to it, so zero classes mapped to the file is valid.

- No Collection classes are mapped to the file.

  Directly mapping collection classes to a partitioned file is not supported, but collection partitioning can be achieved by partitioning the parent class.
Empty Database File

If the database file is empty, the required database file structures to support partitioning can be created in two ways.

- Calling the setPartitioned method on the schema-defined DbFile instance associated with the database file, as shown in the following example.

```java
begin
    beginTransaction;
    Customer.getDbFile.setPartitioned(true);
    commitTransaction;
end;
```

- Executing the JADE Database Administration utility (jdbadmin) with the action=MakePartitioned action, as shown in the following example.

```bash
jdbadmin path=c:\jade\system ini=c:\jade\jade.ini action=MakePartitioned file=customer
```

When partitioning is enabled on a new file, the required partition structures are created when the file is first instantiated. Partitioning can be enabled on an existing empty non-partitioned file, in which case the non-partitioned file is deleted first.

Non-Empty Database File

If the database file is not empty, the required database file structures to support partitioning can be created only by executing the JADE Database Administration utility (jdbadmin) utility with the action=MakePartitioned action, as shown in the following example.

```bash
jdbadmin path=c:\jade\system ini=c:\jade\jade.ini action=MakePartitioned file=customer
```

Setting the partitioned property on an existing map file requires a special file-based reorganization to convert the standard map file to a partitioned file.

Flexibility in Partitioning Database Files

The value of the partitioned attribute is maintained in the database control file, to enable the setting to vary in different instances of the database. This means that when a JADE application is deployed to different sites, some installations could have a selection of database files partitioned while others have a different selection or no partitioned files.

With such flexibility, there is the potential for consistency issues at deployment time if an evolved schema were to violate the file partitioning rules. To help avoid such issues, you can set a partitionable attribute for a DbFile instance, to indicate that the associated database map file is a candidate for partitioning.

Partition Location

When a partition directory is not specified, the database path (that is, the location of the database control file _control.dat) is used. Relative paths are assumed to be relative to the database path.

You can also specify the location of individual partitions. The location of a partition is stored in the partition control record associated with the partition.
Partitioning Database Files

JADE Database Administration utility (jdbadmin) actions enable you to change an offline partition or move an online partition to a different file system location or to change or clear partition locations in a deployed system. For details, see Chapter 2 of the JADE Database Administration Guide.

RootSchema classes (for example, the DbFile and JadeDbFilePartition classes) implement methods that enable you to query or manipulate partition state such as location, including moving the partition to a different file system path, if required.

Partitioning by Period

It is convenient and efficient if data can be partitioned based on the time it is first created. Examples of the kind of data that are candidates for this style of partitioning are application audit trails, transactions, messages, and events.

The objects used to represent these kinds of data are typically created once and are never (or seldom) updated. This kind of data can be partitioned based on the period in which it was created.

To implement a temporal partitioning scheme

1. Select a suitable period for partitioning (daily, monthly, quarterly, or yearly).
2. Enable partitioning on the map file of the class (with the modulus set to 1).
3. Do not re-implement the autoPartitionIndex method.
4. Call the setPartitionIndex method to explicitly set the partition index to zero (0).
5. Implement a mechanism that creates a new partition at the start of each new period.

Note Step 4 of this instruction avoids the overhead of invoking the autoPartitionIndex method defined on the Object class during transaction commit processing (see "Automatic Partitioning", later in this chapter).

With partition modulus set to 1, the creation window is restricted to a single partition (the latest partition), which means that all new objects can be created only in the latest partition. Attempts to set a partition index other than zero (0) will fail, resulting in the aborting of the transaction.

Immediately prior to the commencement of a new period, the application creates new partitions for related classes.

The following code fragment shows the instructions required to create new partitions for the Order and OrderItem classes.

```java
// execute just before midnight at end of current period
beginTransaction;
Order.createPartition;
OrderItem.createPartition;
// execute just after midnight at start of next period
commitTransaction;
```

In this code fragment, the createPartition method creates a new partition for the class and returns the unique partition identifier. The createPartition operation is audited within a database transaction, ensuring that it is atomic and recoverable.

You can make multiple related createPartition operations atomic, by containing them in the same database transaction.
Chapter 20  Partitioning Database Files

The following restrictions apply to the use of the `createPartition` method.

- Partitions can only be created within a transaction
- No other partition creation operation can be in progress
- Persistent objects cannot be created or updated in the transaction that creates a partition
- Persistent objects cannot be created in a partitioned file by any user while a new partition for that file is being created

**Note** For a production application, developers should implement a synchronization mechanism to prevent the creation of objects stored in a partitioned file while a new partition is created.

**Automatic Partitioning**

Automatic partitioning is a strategy where the file partition in which an object is stored is determined within the transaction that creates it, using object state.

A user-defined partitioning scheme, which is essentially an algorithm coded in a partition method, determines the partition for an object. (For more details, see "partitionMethod Option", in Chapter 1.)

To implement automatic partitioning for a class

1. Enable partitioning on the map file of the class.
2. Implement a partitioning algorithm.

**Implementing a Partition Algorithm**

The ways in which an application can control the partition in which a created object is stored are as follows.

- To select the partition using object attributes, define a method that has the `partitionMethod` option specified.
- Explicitly, by calling the `Object` class `setPartitionIndex` method.

**Partition Methods**

You can codify a partitioning scheme for instances of a partitioned class by defining a `partition` method. Partition methods have two main usage contexts, as follows.

- Enabling an application to determine the partition in which an object is stored when the transaction that created it commits.
- Converting a non-partitioned database file containing instances of a single class to partitioned format. (This is achieved by executing a `MakePartitioned` action.)

In the first case, you can reimplement the `autoPartitionIndex` method defined in the `Object` class. If it is not reimplemented and the partition index has not been explicitly set when an object is created, the object is located in the default (latest) partition. In the second case, the name of the partition method is optionally specified for the `MakePartitioned` operation carried out by the JADE Database Administration utility (`jdbadmin`). If no method is specified, the `MakePartitioned` operation invokes the `autoPartitionIndex` method.

A partition method enables you to implement partitioning algorithms based on the values of one or more attributes of the object being created.
Two common single-attribute partitioning strategies are:

- List (or value) partitioning
- Range partitioning

With list partitioning, there is a one-to-one correspondence between an attribute value and a partition. An example might be partitioning sales by region, where the region attribute determines the partition.

With range partitioning, a range of values are mapped to a partition. A simple example might be to use the month part of a date or timestamp attribute to map sales data to one of 12 monthly sales partitions.

The partition method is evaluated during the execution of the `commitTransaction` instruction for each object created in the transaction with a zero partition index. When the returned value is outside the partition modulus range, an exception is raised.

Changes to the property values used in a partition method that occur after the creating transaction commits do not trigger the partition method or cause the object to be relocated.

For more details about partition methods, see "partitionMethod Option", in Chapter 1.

**Determining the Partition Index**

Within a transaction that creates instances of a partitioned class:

- When an object is created, it is initialized with a special partition index value of `not-set`.
- Application logic can call the `setPartitionIndex` method of the `Object` class explicitly, to set the partition index for an object at any time before committing the transaction.
- Application logic can call the `setPartitionID` method of the `Object` class explicitly, to set the absolute partition ID for an object at any time before committing the transaction.

When the transaction commits:

- If the partition index has not been set by the application, the value returned by the `autoPartitionIndex` method is zero (0).
- The value of the partition index determines the partition within the creation window range in which an object is stored.
- Objects with a partition index of zero (0) are stored in the default partition; that is, the latest partition.

**Manual Partitioning**

Manual partitioning covers special case scenarios that cannot be accomplished using the point-in-time or automatic partitioning mechanisms. Manual partitioning could involve using the `createPartition` method in the `Class` class to create new empty partitions and the `moveToPartition` method of the `Object` class to move objects to a specified partition.

This style of partitioning, where objects are relocated individually, is not as efficient as the strategies that allow objects to be located in the correct partition when created. However, it may be necessary to relocate objects between partitions for maintenance or corrective actions, or when the status of an object changes to an inactive state, for example.
Bulk Removal of Objects

You can use the JADE Database Administration utility (jdbadmin) PurgePartition action to efficiently delete the objects stored in a specified partition.

The main purpose of the PurgePartition action is to ensure that inversed references in the active model no longer refer to objects in the partition, so that the partition can be removed.

The PurgePartition action provides a mechanism that efficiently deletes all of the objects (including subobjects) stored in the partition as a bulk operation, in order to maintain referential integrity. The end result of executing a purge is equivalent to deleting each object in the partition without incurring the overhead of deleting the objects individually.

All of the database-level work performed to delete an object in the partition being purged, including file maintenance and auditing, will be skipped entirely.

Conversely, inverse maintenance operations that occur as side-effects of the deletes that affect active objects will be processed and audited normally.

RootSchema Methods Related to Database Partitioning

This section summarizes the JadeDbFilePartition class and the partition-related properties and methods defined in other RootSchema classes. For details, see Volume 1 and Volume 2 of the JADE Encyclopaedia of Classes.

Class Methods

The partition-related methods provided in the Class class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>allInstancesInPartition</td>
<td>Fills an array with instances of the receiver class stored in the specified database file partition</td>
</tr>
<tr>
<td>createPartition</td>
<td>Creates a new empty partition and returns the partition identifier</td>
</tr>
<tr>
<td>getDbFile</td>
<td>Returns the database file to which the class is mapped</td>
</tr>
</tbody>
</table>

DbFile Property and Methods

The partition-related property provided in the DbFile class is summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>partitionable</td>
<td>Specifies whether the DbFile instance can be partitioned (for example, to enforce partition file restrictions in a test or development system where the file is not physically partitioned)</td>
</tr>
</tbody>
</table>

The partition-related methods provided in the DbFile class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>beginPartitionedFileBackup</td>
<td>Begins the backup of selected database file partitions (backs up the partition control file and the partition index file)</td>
</tr>
<tr>
<td>createPartition</td>
<td>Creates a new empty partition and returns the partition identifier</td>
</tr>
</tbody>
</table>
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>endPartitionedFileBackup</td>
<td>Ends the backup of selected database file partitions</td>
</tr>
<tr>
<td>freeze</td>
<td>Converts a file to read-only mode, after which no object update, delete, or create action is allowed</td>
</tr>
<tr>
<td>getOpenPartitions</td>
<td>Populates the input <code>partitionList</code> array with references to <code>JadeDbFilePartition</code> instances</td>
</tr>
<tr>
<td>getPartition</td>
<td>Returns the <code>JadeDbFilePartition</code> instance associated with the indexed partition</td>
</tr>
<tr>
<td>getPartitionCount</td>
<td>Returns the number of non-removed partitions assigned to the file</td>
</tr>
<tr>
<td>getPartitionModulus</td>
<td>Returns the number of partitions in which new instances are stored</td>
</tr>
<tr>
<td>getPartitions</td>
<td>Populates the input <code>partitionList</code> array with <code>JadeDbFilePartition</code> instances</td>
</tr>
<tr>
<td>isFrozen</td>
<td>Returns <code>true</code> if the database file has been frozen</td>
</tr>
<tr>
<td>isPartitioned</td>
<td>Returns <code>true</code> if the database file is partitioned</td>
</tr>
<tr>
<td>setPartitionModulus</td>
<td>Specifies the number of database partitions in which new instances are stored</td>
</tr>
<tr>
<td>setPartitioned</td>
<td>Changes the partitioned attribute of an empty (non-instantiated) database file</td>
</tr>
<tr>
<td>thaw</td>
<td>Restores the database file to its default active state</td>
</tr>
</tbody>
</table>

### JadeDbFilePartition Properties and Methods

The partition-related properties provided in the `JadeDbFilePartition` class are summarized in the following table.

<table>
<thead>
<tr>
<th>Property</th>
<th>Contains…</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dbFile</code></td>
<td>A reference to the parent <code>DbFile</code> instance</td>
</tr>
<tr>
<td><code>partitionID</code></td>
<td>The partition identifier assigned to the partition</td>
</tr>
</tbody>
</table>

The partition-related methods provided in the `JadeDbFilePartition` class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>allInstances</code></td>
<td>Populates the specified object array specified with references to objects stored in the partition associated with the receiver</td>
</tr>
<tr>
<td><code>backupFilePartition</code></td>
<td>Backs up a single partition of a physical database file</td>
</tr>
<tr>
<td><code>certifyFile</code></td>
<td>Initiates the certification of a single database file partition</td>
</tr>
<tr>
<td><code>display</code></td>
<td>Returns details about a single database file partition</td>
</tr>
<tr>
<td><code>freeze</code></td>
<td>Converts a partition to read-only mode, after which no object update, delete, or create is permitted</td>
</tr>
<tr>
<td><code>getBackupTimestamp</code></td>
<td>Returns a timestamp containing the date and time the database file partition was last backed up</td>
</tr>
<tr>
<td><code>getCreationTimestamp</code></td>
<td>Returns a timestamp containing the date and time the database file partition was created</td>
</tr>
<tr>
<td><code>getFileLength</code></td>
<td>Returns the size of a single physical database file partition</td>
</tr>
</tbody>
</table>
### Method Description

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getFileStatus</td>
<td>Returns the status of a single physical database file partition during the backup process</td>
</tr>
<tr>
<td>getFreeSpace</td>
<td>Evaluates the available free-space in a single database file partition</td>
</tr>
<tr>
<td>getFullBackupTimestamp</td>
<td>Returns a timestamp containing the date and time the database file was last backed up</td>
</tr>
<tr>
<td>getLabel</td>
<td>Returns the label assigned to the partition</td>
</tr>
<tr>
<td>getLocation</td>
<td>Returns the file system path assigned to the partition</td>
</tr>
<tr>
<td>getModifiedTimestamp</td>
<td>Returns a timestamp containing the date and time the database file partition was last updated</td>
</tr>
<tr>
<td>getName</td>
<td>Returns the name of the database file partition</td>
</tr>
<tr>
<td>getStatistics</td>
<td>Returns statistics on reads of single database partition activity</td>
</tr>
<tr>
<td>isFrozen</td>
<td>Returns <code>true</code> if the associated partition has been frozen</td>
</tr>
<tr>
<td>isOffline</td>
<td>Returns <code>true</code> if the associated partition has been marked as offline</td>
</tr>
<tr>
<td>markOffline</td>
<td>Marks a partition as officially absent so that it can be taken offline</td>
</tr>
<tr>
<td>markOnline</td>
<td>Marks a partition as present after it has been brought back online</td>
</tr>
<tr>
<td>move</td>
<td>Changes the location attribute and moves the partition to the specified destination</td>
</tr>
<tr>
<td>setLabel</td>
<td>Assigns or changes the logical label associated with the partition</td>
</tr>
<tr>
<td>setLocation</td>
<td>Changes the default or designated physical location of a database file partition</td>
</tr>
<tr>
<td>thaw</td>
<td>Restores the database file partition to its default active state</td>
</tr>
</tbody>
</table>

### Object Methods

The partition-related methods provided in the `Object` class are summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>autoPartitionIndex</td>
<td>Returns the partition index of the database file partition in which the receiver is stored on creation</td>
</tr>
<tr>
<td>moveToPartition</td>
<td>Moves the receiver object to the specified partition</td>
</tr>
<tr>
<td>setPartitionIndex</td>
<td>Specifies the partition in which to locate the receiver</td>
</tr>
</tbody>
</table>

### System Method

The partition-related method provided in the `System` class is summarized in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getObjectPartitionID</td>
<td>Returns the identifier of the partition in which the specified object is located</td>
</tr>
</tbody>
</table>
Batch JADE Database Administration Utility Partitioning Actions

This section summarizes the JADE Database Administration utility (jdbadmin) commands relating to the partitioning of database files and the maintenance of the partitions. For details, see "Using the JADE Database Administration Utility", in Chapter 2 of the JADE Database Administration Guide.

The partition-related values for the action argument when running the JADE Database Administration utility are summarized in the following table.

<table>
<thead>
<tr>
<th>Action Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Backup</td>
<td>Initiates a full backup that includes all online database files or a partial backup that excludes frozen files or frozen partitions, or both frozen files and partitions</td>
</tr>
<tr>
<td>Certify</td>
<td>Performs a read-only check on specified database files and database partitions</td>
</tr>
<tr>
<td>FreeSpace</td>
<td>Evaluates the free-space in specified database files and database partitions</td>
</tr>
<tr>
<td>Freeze</td>
<td>Freezes schema files, specified database files, and database partitions</td>
</tr>
<tr>
<td>FreezeSchemaFiles</td>
<td>Changes the volatility of the _userint, _userscm, and _userxrf schema files to frozen</td>
</tr>
<tr>
<td>ListDbFiles</td>
<td>Displays all schema-defined DbFile entities and their attributes</td>
</tr>
<tr>
<td>MakePartitioned</td>
<td>Partitions a specified database file</td>
</tr>
<tr>
<td>MarkOffline</td>
<td>Change the state of frozen database files, and database partitions before taking offline</td>
</tr>
<tr>
<td>MarkOnline</td>
<td>Change the state of frozen database files, and database partitions before bringing online</td>
</tr>
<tr>
<td>MovePartition</td>
<td>Moves a database partition to a specified location</td>
</tr>
<tr>
<td>PurgePartition</td>
<td>Removes inverse references to objects stored in the specified online or offline database partition</td>
</tr>
<tr>
<td>RemovePartition</td>
<td>Remove the specified empty database partition</td>
</tr>
<tr>
<td>SetFilePath</td>
<td>Sets control record paths for specified database files and database partitions</td>
</tr>
<tr>
<td>SetPartitionLabel</td>
<td>Changes the label of the specified database partition</td>
</tr>
<tr>
<td>SetPartitionLocation</td>
<td>Changes the location of the specified database partition</td>
</tr>
<tr>
<td>Thaw</td>
<td>Thaws schema files, specified database files, and database partitions</td>
</tr>
<tr>
<td>ThawSchemaFiles</td>
<td>Changes the volatility of frozen _userint, _userscm, and _userxrf schema files to transparent</td>
</tr>
</tbody>
</table>
Chapter 21  Runtime Data Definition

This chapter contains the following topics.

- Overview
- Runtime Dynamic Properties
  - Dynamic Property Clusters
    - Creating and Maintaining a Dynamic Property Cluster
  - Runtime Dynamic Properties
    - Adding a Runtime Dynamic Property
    - Changing a Runtime Dynamic Property
    - Maintaining the Value of a Runtime Dynamic Property
    - Deleting a Runtime Dynamic Property
    - Inverse References Involving Runtime Dynamic Properties
    - Runtime Dynamic Property Restrictions
- Adding User Classes at Run Time
- Dynamic Objects
- DynaDictionary Objects

Overview

This chapter covers features that relate to dynamically defining data at run time: dynamic properties, dynamic objects, dynamic dictionary objects, and user classes.

Dynamic properties provide an efficient and easy way to add properties to existing persistent instances without requiring a database reorganization. They are also convenient for situations when ad hoc values need to be attached to existing classes. In addition, changing or deleting dynamic properties does not require a reorganization.

The two types of dynamic property are:

- Runtime dynamic properties, which are added through code when an application is running.
  
  For details, see "Runtime Dynamic Properties", later in this chapter.

- Design-time dynamic properties, which are added through the Class Browser when designing classes.
  
  For details, see "Dynamic Clusters and Properties", in Chapter 4 of the JADE Development Environment User's Guide.

Dynamic objects are instances of the JadeDynamicObject class. They are self-describing objects whose attributes are specified and available at run time.

Dynamic dictionaries, which are instances of the DynaDictionary class, enable you to access entries in member key dictionary subclasses but to defer the specification of the membership and keys until run time.
User classes are class definitions that are added at runtime by a user application.

Runtime Dynamic Properties

Runtime dynamic properties:

- Are created, changed, or deleted at runtime by user logic only.
- Do not require any reorganization when added, changed, or deleted.
- Are displayed in the JADE development environment for a class (the default text color is magenta) but cannot be changed or deleted using the development environment.
- Are not included in schema (.scm) files with the class.
- Cannot be directly referred to in JADE methods. Access is by calls to the `getPropertyValue` and `setPropertyValue` methods using the property name.
- Reorganization is not available to handle any impact caused; for example, an added inverse is not populated.
- Cannot be used in exposures, ODBC, packages, RPS, as keys, in an inverse with a static or design-time property on the other side, JADE audit access, or a `delete` command action in a JADE Control File (JCF) `commandFile` argument in the batch Schema Load utility (`jadloadb`).

For details about creating and maintaining a design-time dynamic property, see "Defining a Dynamic Attribute Property" and "Defining a Dynamic Reference Property", in Chapter 4 of the JADE Development Environment User’s Guide.

Dynamic Property Clusters

Dynamic properties are stored in separate storage units, called `dynamic property clusters`.

**Note** This section applies to runtime and design-time dynamic properties.

A cluster is associated with the object, which stores the static properties. For better performance, dynamic properties that are typically accessed together can be assigned to specific clusters.

In the following example, a `Customer` class has a number of static properties; for example, a `name` property which is a string of length 50 characters. Logic has been provided to enable users of the application to add information for contacting a person. A dynamic property cluster called `CustomerCluster01` has been added for this purpose and two dynamic properties called `email` and `cell` have been added to the cluster, as shown in the following diagram.
Another property cluster called CustomerCluster02 has been added, to which a dynamic property called comment has been added.

A dynamic property of type String, StringUtf8, or Binary with a length greater than 540 is stored as a non-embedded slob, slobutf8, or blob, respectively. The maximum length of an embedded Binary type is 540 characters, an embedded StringUtf8 type is 540 characters, and an embedded String type is 539 characters.

The isDesignTimeDynamicProperty and isRunTimeDynamicProperty methods in the Property class can be used to distinguish between design-time and runtime dynamic properties, as shown in the following example.

```plaintext
vars
  prop: Property;
begin
  foreach prop in Customer.allPropertiesUpTo(Object) do
    if prop.isDesignTimeDynamicProperty then
      write prop.name & " is a design-time dynamic property";
    elseif prop.isRunTimeDynamicProperty then
      write prop.name & " is a runtime dynamic property";
    else
      write prop.name & " is a static property";
    endif;
  end;
end;
```

### Creating and Maintaining a Dynamic Property Cluster

You can create and maintain dynamic property clusters for a class in the Class Browser, provided it is not a collection class. For more details, see "Defining Dynamic Property Clusters", in Chapter 3 of the JADE Development Environment User's Guide.

Alternatively, you can create and maintain dynamic property clusters by executing the methods of the Class class shown in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addDynamicPropertyCluster</td>
<td>Adds and returns a new dynamic property cluster to the receiving class</td>
</tr>
<tr>
<td>compactDynamicPropertyClusters</td>
<td>Compacts dynamic property clusters in which dynamic property instances were deleted</td>
</tr>
<tr>
<td>deleteDynamicPropertyCluster</td>
<td>Deletes a dynamic property cluster from the receiving class</td>
</tr>
<tr>
<td>findDynamicPropertyCluster</td>
<td>Returns the dynamic property cluster with the specified name from the class of the receiver</td>
</tr>
</tbody>
</table>
In the following example, a runtime dynamic property cluster called `CustomerCluster` is added to the `Customer` class.

```java
begin
  beginTransaction;
  Customer.addDynamicPropertyCluster("CustomerCluster");
  commitTransaction;
end;
```

### Runtime Dynamic Properties

Runtime dynamic properties are added and maintained through the execution of application code. Although they are displayed in the Properties List window for a class (the default text color is magenta), you cannot add, delete, or change a runtime dynamic property in the Class Browser.

If the class is versioned, runtime dynamic properties and clusters can be added to the latest version only. If production mode is set, dynamic properties and clusters can be changed or deleted in single user mode only.

### Adding a Runtime Dynamic Property

You can add a runtime dynamic property by using the following methods of the `JadeDynamicPropertyCluster` class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addDynamicProperty</code></td>
<td>Adds a new runtime dynamic property to the receiving cluster using the specified information.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> The <code>addDynamicProperty</code> method of the <code>Class</code> class provides equivalent functionality.</td>
</tr>
<tr>
<td><code>addExclusiveDynamicProperty</code></td>
<td>Adds a new exclusive runtime dynamic property to the receiving cluster using the specified information.</td>
</tr>
</tbody>
</table>

In the following example, a runtime dynamic property called `email` is added to the `Customer` class into a cluster called `CustomerCluster` that must have been created previously.

```java
vars
  cluster : JadeDynamicPropertyCluster;
begin
  cluster := Customer.findDynamicPropertyCluster("CustomerCluster");
  beginTransaction;
  cluster.addDynamicProperty("email", String, 30, 0.Byte);
  commitTransaction;
end;
```

The previous example could be coded with the following equivalent.

```java
begin
  beginTransaction;
  Customer.addDynamicProperty("CustomerCluster", "email", String, 30, 0.Byte);
  commitTransaction;
end;
```
In the following example, an exclusive runtime dynamic property called allAppointments is added to the Customer class.

```javascript
vars
  cluster : JadeDynamicPropertyCluster;
begin
  cluster := Customer.findDynamicPropertyCluster("CustomerCluster");
  beginTransaction;
  cluster.addExclusiveDynamicProperty("allAppointments", AppointmentDict);
  commitTransaction;
end;
```

**Note** You cannot add a runtime dynamic property through the Class Browser.

## Changing a Runtime Dynamic Property

You can use the changeDynamicProperty method of the Property class to change the name, type, and length of a runtime dynamic property. The method has the following signature.

```javascript
changeDynamicProperty(propertyName: String;
  propertyType: Type;
  length: Integer;
  scaleFactor: Byte) updating;
```

You can change any or all of the `propertyName`, `propertyType`, `length`, and `scaleFactor` parameter values. The `length` parameter is required for Binary, String, and StringUtf8 primitive types. The `scaleFactor` parameter is required for Decimal primitive types.

The type of a primitive property cannot be changed to a reference, and the reverse.

No reorganization is required when you change the type of a runtime dynamic property. If the actual value of a runtime dynamic property differs from the type of the property definition, the value is converted when the value is retrieved. Dynamic property values are always stored using the type of the property definition when the value is updated.

If you reduce the length of a Binary, String, or StringUtf8 runtime dynamic property, any populated values that are longer than the new length are truncated when the property is fetched. The new length is enforced when a new value is assigned to the property.

**Notes** You cannot change a runtime dynamic property if the class in which it is defined is being used by another process or there is an instance of this class or any subclass.

You cannot reduce the length of a non-embedded slob, slobUtf8, or blob dynamic property to less than 540 characters or bytes. The maximum length of an embedded Binary type is 540 characters, an embedded StringUtf8 type is 540 characters, and an embedded String type is 539 characters.

You can use the changeExclusiveDynamicProperty method of the Property class to change the name of an exclusive runtime dynamic property; for example, an exclusive collection reference. The method has the following signature.

```javascript
changeExclusiveDynamicProperty(propertyName: String) updating;
```

**Note** You cannot change runtime dynamic properties through the Class Browser.
Maintaining the Value of a Runtime Dynamic Property

You can maintain the value of a runtime dynamic property by using the following methods of the `Object` class.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>deletePropertyValue</code></td>
<td>Sets the value of the property specified by the <code>name</code> parameter to null</td>
</tr>
<tr>
<td><code>getPropertyValue</code></td>
<td>Returns the value of the property specified by the <code>name</code> parameter</td>
</tr>
<tr>
<td><code>setPropertyValue</code></td>
<td>Sets the property of the receiver to the specified value</td>
</tr>
<tr>
<td><code>tryGetPropertyValue</code></td>
<td>Returns the value of the specified property, if it exists; otherwise it returns <code>false</code></td>
</tr>
</tbody>
</table>

In the following example, the value of a runtime dynamic property, called `email`, is set to `wilbur@somewhere.com` for an instance of the `Customer` class.

```plaintext
vars
cust : Customer;
begin
  beginTransaction;
  create cust;
  cust.setPropertyValue("email","wilbur@somewhere.com");
  commitTransaction;
end;
```

**Note** You cannot access a runtime dynamic property directly by name in a JADE method; for example, the following instruction would not compile.

```plaintext
cust.email := "wilbur@somewhere.com";
```

Deleting a Runtime Dynamic Property

You can delete a runtime dynamic property by using the `deleteDynamicProperty` method of the `JadeDynamicPropertyCluster` class or the equivalent `deleteDynamicProperty` method of the `Class` class.

In the following example, a runtime dynamic property called `email` is deleted from the `Customer` class.

```plaintext
vars
  cluster : JadeDynamicPropertyCluster;
begin
  cluster := Customer.findDynamicPropertyCluster("CustomerCluster");
  beginTransaction;
  cluster.deleteDynamicProperty("email");
  commitTransaction;
end;
```

You could also code the previous example as follows.

```plaintext
begin
  beginTransaction;
  Customer.deleteDynamicProperty("email");
  commitTransaction;
end;
```
Note You can delete a runtime dynamic property only if the class in which it is defined is not being used by any other process. If production mode is set, you can delete a runtime dynamic property in single user mode only.

Inverse References Involving Runtime Dynamic Properties

JADE inverse references are used to define relationships between classes. For details about using the JADE development environment to define a relationship, see "Defining an Inverse Reference Property", in Chapter 4 of the JADE Development Environment User’s Guide.

A relationship can also be defined using runtime dynamic properties as the inverse references. For example, a one-to-many relationship (one customer has many appointments) could be established between the Customer class and Appointment class, as shown in the following diagram.

![Diagram showing relationship between Customer and Appointment classes]

The runtime dynamic properties implementing the relationship are allAppointments, which is an exclusive collection property in a dynamic property cluster associated with the Customer class, and myCustomer, which is a property in a dynamic property cluster associated with the Appointment class.

The Property class method addDynamicInverse, which has the following signature, adds an inverse between the receiving dynamic property and the dynamic property specified in the property parameter.

```java
addDynamicInverse(property: Property;
mode: Character;
kind: Character): Property subschemaFinal, updating, abstract;
```

This method returns a reference to the receiving dynamic property with the inverse added. Note that when the first inverse is added, this will be different from the receiver.

The property parameter must identify a compatible dynamic property; for example, if the type of the property is a Collection class, the schema type of the receiver must be compatible with the membership of the collection.

Note If instances of the class of either dynamic property exist and the dynamic properties have non-null values, the inverse relationship is not validated or populated when the inverse definition is added.

The values for the mode parameter are defined by the following ExplicitInverseRef class constants.

- UpdateMode_Automatic
- UpdateMode_ManAuto
- UpdateMode_Manual
The values for the `kind` parameter are defined by the following `ExplicitInverseRef` class constants.

- `Kind_Child`
- `Kind_Parent`
- `Kind_Peer`

The `Property` class `removeDynamicInverse` method, which has the following signature, removes an inverse reference between the dynamic property specified in the `property` parameter and the receiver.

```java
removeDynamicInverse(property: Property): Property subschemaFinal, updating, abstract;
```

This method returns a reference to the receiving dynamic property with the inverse removed. Note that when the last inverse is removed, this will be different from the receiver.

In the following example, the runtime dynamic properties for each class are created, and then an inverse relationship is established between them.

```java
vars
custCluster, appCluster : JadeDynamicPropertyCluster;
myCustomer, allAppointments : Property;
begin
  beginTransaction;
    // Add 'allAppointments'
custCluster := Customer.findByIdDynamicPropertyCluster("CustomerCluster");
allAppointments := custCluster.addExclusiveDynamicProperty("allAppointments",
                                  AppointmentDict);
    // Add 'myCustomer'
appCluster := Appointment.findByIdDynamicPropertyCluster("AppointmentCluster");
myCustomer := appCluster.addDynamicProperty("myCustomer", Customer, 0, 0.Byte);
    // Define inverse relationship
myCustomer := myCustomer.addDynamicInverse(allAppointments,
                                  ExplicitInverseRef.UpdateMode_Manual,
                                  ExplicitInverseRef.Kind_Peer);
  commitTransaction;
end;
```

**Notes**: Code that sets up inverse references must be executed before code that uses the inverses.

You cannot define inverses between runtime dynamic properties through the Class Browser.

### Runtime Dynamic Property Restrictions

You cannot:

- Add more than 65,535 dynamic property clusters to a class.
- Add a runtime dynamic property to a system class.
- Add a runtime dynamic property to a `Collection` class or subclass.
- Delete or change a runtime dynamic property selected in the Properties List of the Class Browser. (You can do this only from the `Property` class `changeDynamicProperty` method or the `JadeDynamicPropertyCluster` class `deleteDynamicProperty` method.)
Use runtime dynamic properties by name in JADE methods. Exception 1068 (Feature not available) is raised if a runtime dynamic property is accessed directly (by name) in a method. (Access a runtime dynamic property with the Object class getPropertyValue and setPropertyValue methods.)

Select a runtime dynamic property from the list of properties in the following places.

- Dictionary key
- Key path of a dictionary
- Properties on a class exported in a package
- C# or Web service exposures
- Design-time property in the JADE Painter
- Relational view
- RPS mapping

Add a dynamic property cluster to a subclass of the Control class.

Deploy runtime dynamic property changes for a schema from a schema extract. (Runtime dynamic properties are not included when you extract a schema.)

Rename or delete a runtime dynamic property by using a JADE command file in the batch Schema Load utility.

Reduce the length of a non-embedded Binary and StringUtf8 runtime dynamic property that is longer than 540 to fewer than 540 bytes or characters.

Reduce the length of a non-embedded String runtime dynamic property that is longer than 539 to fewer than 539 bytes or characters.

Delete a Binary, String, and StringUtf8 runtime dynamic property if instances of the class exist.

**Note** Changing runtime dynamic properties and clusters of a class in the same transaction in which an instance of the class is created can result in exception 3158 (Partition method execution failed) being raised on the server in multiuser mode, if the database file is partitioned and the class is not open.

You should use one transaction to change runtime dynamic property metadata and a separate transaction for object updates.

## Adding User Classes at Run Time

Your user applications can add user classes at run time. Although these user-defined classes are visible in the JADE development environment, they are not considered as part of the JADE model, you cannot reference them in JADE methods, they are not extracted, reorganized, and so on.

You can only define runtime dynamic properties on user classes. You can create instances of a user class of any lifetime (that is, persistent, transient, and shared-transient lifetimes).

The following table summarizes the classes and methods that enable you to maintain user classes.

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schema</td>
<td>addUserCollectionSubclass</td>
<td>Creates a user collection class as a subclass of the specified superclass in the receiving schema</td>
</tr>
</tbody>
</table>
The following example shows a runtime dynamic property being added to a user class.

```java
vars
  userClass1 : Class;
  cluster : JadeDynamicPropertyCluster;
  c1 : C1;
begin
  beginTransaction;
  userClass1 := currentSchema.addUserSubclass(C1, "UserClass1", "MapFileName");
  cluster := userClass1.addDynamicPropertyCluster("Cluster");
  cluster.addDynamicProperty("dynamicInteger", Integer, 0, 0.Byte);
  commitTransaction;
  create c1 as userClass1 transient;
  c1.integer := 123; // static property
  c1.setPropertyValue("dynamicInteger", 456); // runtime dynamic property
  c1.inspectModal();
  delete c1;
end;
```

### Dynamic Objects

The **JadeDynamicObject** class implements the structure and behavior of dynamic objects. A JADE dynamic object is a self-describing object whose attributes are determined at run time.

The **JadeDynamicObject** class has two fixed attributes: **type** and **name**, which you can use to determine the runtime type of a dynamic object. The JADE Inspector displays the type, name, and dynamic attribute names and values of **JadeDynamicObject** instances.

In the following example, a dynamic object is created and the three properties called **Item**, **Quantity**, and **Available** are added and their values are set. At the end of the method, the names and values of the properties are displayed.

```java
vars
  jdo : JadeDynamicObject;
  i : Integer;
  str : String;
begin
  create jdo;
  jdo.setItem("Item", "toy");
  jdo.setQuantity("Quantity", 12);
  jdo.setAvailable("Available", true);
  jdo.inspectModal();
  delete jdo;
end;
```
jdo.addProperty("Item", String);
jdo.setPropertyValue("Item", "calculator");
jdo.addProperty("Quantity", Integer);
jdo.setPropertyValue("Quantity", 100);
jdo.addProperty("Available", Boolean);
jdo.setPropertyValue("Available", true);
foreach i in 1 to jdo.propertyCount do
    str := str & CrLf & jdo.getPropertyName(i) & " = " & jdo.getPropertyValueByIndex(i).String;
endforeach;
write str;
epilog
    delete jdo;
end;

The output from the previous method is as follows.

Item = calculator
Quantity = 100
Available = true

The method in the following example creates a dynamic object and passes this to the Collection class getStatistics method, which populates the object with various statistical attributes and their values, returning the dynamic object to the caller. The method then uses the getPropertyName, getPropertyValueByIndex, and propertyCount methods to display the statistical attribute name and value pairs. As the calling method created the jdo variable, it is also responsible for deletion, which is performed in an epilog.

vars
    jdo : JadeDynamicObject;
    str : String;
    int : Integer;
    count : Integer;
begin
    create jdo;
    node.processes.getStatistics(jdo);
    str := '---' & jdo.getName & '(' & jdo.type.String & ')---';
    count := jdo.propertyCount;
    foreach int in 1 to count do
        str := str & CrLf & jdo.getPropertyName(int) & " = " & jdo.getPropertyValueByIndex(int).String;
    endforeach;
    write str;
    epilog
        delete jdo;
end;

For details about the properties and methods defined in the JadeDynamicObject class, see "JadeDynamicObject Properties" and "JadeDynamicObject Methods", in Chapter 1 of the JADE Encyclopaedia of Classes.

DynaDictionary Objects

The DynaDictionary class enables you to access entries in member key dictionary subclasses but to defer the specification of the membership and keys until run time.
Dynamic dictionaries are useful in applications with requirements for:

- Ad hoc queries or collection-based sorts without the overhead of maintaining multiple persistent dictionaries.
- Intensive collation or collection-based sorting. The sorting provided by a dynamic dictionary is sometimes referred to as an insertion sort, in which each entry is inserted in the correct place in the structure as opposed to moving the entries around to obtain the required order.

As with any collection, the size of a dynamic dictionary is limited by the maximum entries for a collection (2^32) or the available disk space provided for the transient database.

**Note**  Dynamic dictionaries do not offer a facility to sort objects not entirely based on a comparison of embedded attribute values; for example, the ability for you to provide your own sort compare routine is not supported.

In the following example, a dynamic dictionary performs an object sort based on member attributes. The `publications` property is a collection of `Publications`. This example shows the use of the reimplemented `Dictionary` class `startKeyGeq` method to start the iteration at a specific point and demonstrates passing a variable list of key parameters where the keys are not known at compile-time.

```plaintext
vars
dynaDict : DynaDictionary;
pub : Publication;
iter : Iterator;
begin
create dynaDict transient;
// set the membership of our dynamic dictionary
dynaDict.setMembership(Publication);
// specify the ytdSales, royalty, and descending pubDate dictionary keys
dynaDict.addMemberKey("ytdSales", false, false);
dynaDict.addMemberKey("royalty", false, false);
// specify descending key so that most recent titles appear first
dynaDict.addMemberKey("pubdate", true, false);
// complete key definition
dynaDict.endKeys(false);
// copy publication instances into the dynamic dictionary
publications.copy(dynaDict);
// display all publications with more than 1000
// sales (ytd) in sorted order
iter := dynaDict.createIterator;
// start the iteration where ytdSales >= 1000
// since the dynadict has 3 keys we must pass
// keys to the startKeyGeq method
dynaDict.startKeyGeq(1000, null, null, iter);
while iter.next(pub) do
  write pub.name & " " & pub.ytdSales.String &
    pub.royalty.String & " " & pub.pubdate.String;
endwhile;
epilog
  // ensure we delete transients
  delete iter;
  delete dynaDict;
end;
```

For details about the methods defined in the `DynaDictionary` class, see "DynaDictionary Methods", in Volume 1 of the JADE Encyclopaedia of Classes. For details about passing variable parameters to methods, see "Passing Variable Parameters to Methods", in Chapter 1 of the JADE Developer's Reference.
Chapter 22 Unaudited Database File Operations

When batch-loading a database file or when generating ad hoc indexes, for example, there is significant auditing overhead. Until the index build is complete, the database file or partition is known to be building (application preserved state) and if the build is restarted, the file or partition is first dropped.

You can update database files and partitions with auditing disabled, to eliminate journal disk space use and I/O overhead when loading data. When auditing is re-enabled for a file or partition, a copy of the file or partition is compressed by default, and inserted into the journal. During database roll-forward or replay, the file at the database location is replaced by the file reconstructed from the journal. Subsequent audited updates therefore replay correctly.

Caution Disable the auditing of database files and partitions only when restart recovery is not required. Unaudited operations cannot be used when you have RPS secondaries, as this would result in SQL inconsistencies.

The following table summarizes the methods used to manage unaudited operations.

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbFile</td>
<td>disableAuditing</td>
<td>Disables the auditing associated with object operations performed against the file</td>
</tr>
<tr>
<td>DbFile</td>
<td>drop</td>
<td>Removes the file and marks it as deleted</td>
</tr>
<tr>
<td>DbFile</td>
<td>enableAuditing</td>
<td>Re-enables the auditing associated with object operations performed against the file</td>
</tr>
<tr>
<td>DbFile</td>
<td>isAuditing</td>
<td>Returns true if auditing associated with object operations performed against the file is enabled and returns false when auditing has been disabled</td>
</tr>
<tr>
<td>JadeDatabaseAdmin</td>
<td>doQuietpoint</td>
<td>Attempts to establish a database quiet point</td>
</tr>
<tr>
<td>JadeDbFilePartition</td>
<td>disableAuditing</td>
<td>Disables auditing associated with object operations performed against the partition</td>
</tr>
<tr>
<td>JadeDbFilePartition</td>
<td>drop</td>
<td>Removes the partition and marks it as deleted</td>
</tr>
<tr>
<td>JadeDbFilePartition</td>
<td>enableAuditing</td>
<td>Re-enables the auditing associated with object operations performed against the partition</td>
</tr>
<tr>
<td>JadeDbFilePartition</td>
<td>isAuditing</td>
<td>Returns true if auditing associated with object operations performed against the partition is enabled and returns false when auditing has been disabled</td>
</tr>
</tbody>
</table>

In addition, the DbFile class provides the EnableAudit_NoCompress class constant. For details, see Volume 1 of the JADE Encyclopaedia of Classes.
## Appendix A

### JADE Limits

This appendix covers the JADE limits, which are listed in the following table.

<table>
<thead>
<tr>
<th>Item</th>
<th>Maximum Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application name</td>
<td>100 characters</td>
</tr>
<tr>
<td>Armed local exception handlers for each called method</td>
<td>128</td>
</tr>
<tr>
<td>Authentication protocol key size</td>
<td>512 bytes</td>
</tr>
<tr>
<td>Binary large objects (blobs) and string large objects</td>
<td>65,535 (combined total)</td>
</tr>
<tr>
<td>(slobs) in a class</td>
<td></td>
</tr>
<tr>
<td>Binary length</td>
<td>2,147,483,647 bytes</td>
</tr>
<tr>
<td>Binary length in a user-defined <strong>BinaryArray</strong></td>
<td>16,000 characters</td>
</tr>
<tr>
<td>Bubble help display</td>
<td>2,000 characters</td>
</tr>
<tr>
<td>Characters displayed in a line of a <strong>JadeTextEdit</strong> control</td>
<td>The first 8190 characters</td>
</tr>
<tr>
<td>Collection entries</td>
<td>$2^{32} - 1$ (4,294,967,295)</td>
</tr>
<tr>
<td>Collection physical block size</td>
<td>256*1024 (262,144)</td>
</tr>
<tr>
<td>Columns in a table control</td>
<td>16,000</td>
</tr>
<tr>
<td>Combo box (collection display)</td>
<td>32,000 items</td>
</tr>
<tr>
<td>Concurrent processes across the entire database</td>
<td>$2^{16} - 1$ (65,535)</td>
</tr>
<tr>
<td>Control caption</td>
<td>32,767 characters</td>
</tr>
<tr>
<td>Control name</td>
<td>86 characters</td>
</tr>
<tr>
<td>Data returned when a string is updated by the <strong>read</strong> instruction</td>
<td>2048 characters</td>
</tr>
<tr>
<td>Database file name and path</td>
<td>260 characters</td>
</tr>
<tr>
<td>Database file size</td>
<td>$2^{64} - 1$ (approximately 16 Exabytes)</td>
</tr>
<tr>
<td>Date – earliest that can be formatted</td>
<td>1 January 1601</td>
</tr>
<tr>
<td>Date – latest that can be formatted</td>
<td>31st December 30827</td>
</tr>
<tr>
<td>Decimal variable maximum precision</td>
<td>23 digits</td>
</tr>
<tr>
<td>Delta name</td>
<td>8 characters</td>
</tr>
<tr>
<td>Dictionary key size (when concatenated)</td>
<td>512 key units</td>
</tr>
<tr>
<td><strong>Note</strong> A key unit is a byte for any non-character data type, or one character for any character data type; that is, key sizes are string-encoding agnostic. Key sizes also must allow for a null character to terminate any strings; characters are not null-terminated.</td>
<td></td>
</tr>
<tr>
<td>Dynamic property clusters in a class</td>
<td>65,535</td>
</tr>
</tbody>
</table>
### JADE Limits

<table>
<thead>
<tr>
<th>Item</th>
<th>Maximum Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event method name</td>
<td>For controls, the concatenation of the property name and method name cannot exceed 100 characters</td>
</tr>
<tr>
<td>Exclusive collections (subobjects) in a class</td>
<td>65,535</td>
</tr>
<tr>
<td>External function and external method entry point</td>
<td>255 characters (with any entry point longer than 100 characters enclosed in single quote or double quote characters)</td>
</tr>
<tr>
<td>File loaded into a <code>JadeTextEdit</code> control</td>
<td>The smaller of 50M bytes or a tenth of the physical memory</td>
</tr>
<tr>
<td>Font name</td>
<td>31 characters</td>
</tr>
<tr>
<td>Form caption</td>
<td>255 characters</td>
</tr>
<tr>
<td>Form name</td>
<td>100 characters</td>
</tr>
<tr>
<td>Height of form or control</td>
<td>32,767 pixels</td>
</tr>
<tr>
<td>Host name</td>
<td>1,024</td>
</tr>
<tr>
<td>Hyperlink string</td>
<td>64,000 characters</td>
</tr>
<tr>
<td>Instances of a class</td>
<td>$2^{64}-1$ (approximately 16 Exabytes)</td>
</tr>
<tr>
<td>Integer variable</td>
<td>-2,147,483,648 through 2,147,483,647</td>
</tr>
<tr>
<td>Integer64 variable</td>
<td>-9,223,372,036,854,775,808 through 9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>Interpreter Output Viewer display</td>
<td>4M bytes of text, truncated to the nearest full line</td>
</tr>
<tr>
<td><code>JadeBytes</code> content retrieved by the <code>getContent</code> method</td>
<td>The smaller of 1G byte or a third of the physical memory</td>
</tr>
<tr>
<td><code>JadeBytes</code> instance</td>
<td>1,019G bytes, approximately</td>
</tr>
<tr>
<td><code>JadeDynamicObject</code> property name</td>
<td>100 characters</td>
</tr>
<tr>
<td>List box (collection display)</td>
<td>32,000 items</td>
</tr>
<tr>
<td>List box level</td>
<td>63 items</td>
</tr>
<tr>
<td>Local variable and local constant identifiers</td>
<td>100 characters</td>
</tr>
<tr>
<td>Map file name</td>
<td>100 characters</td>
</tr>
<tr>
<td>Menu caption</td>
<td>100 characters</td>
</tr>
<tr>
<td>Methods in a class</td>
<td>65,535</td>
</tr>
<tr>
<td>Nested exceptions</td>
<td>20</td>
</tr>
<tr>
<td>Nested levels for read transactions (bracketed by <code>beginLoad / endLoad</code>, and <code>beginLock / endLock</code> instructions)</td>
<td>255</td>
</tr>
<tr>
<td>Network address (IPv6)</td>
<td>65</td>
</tr>
<tr>
<td>Notifications containing binary or string (<code>Binary</code>, <code>String</code>, <code>StringUtf8</code>) data</td>
<td>48K bytes when data is sent across the network; 2G bytes for applications running within the server node in single user and server applications only</td>
</tr>
<tr>
<td>Item</td>
<td>Maximum Limit</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Parameters in a JADE method, external method, or external function</td>
<td>129</td>
</tr>
<tr>
<td>Partitions in a database file</td>
<td>$2^{32}$ (4,294,967,296)</td>
</tr>
<tr>
<td>Password length</td>
<td>30 characters</td>
</tr>
<tr>
<td>Pipe name string</td>
<td>256 characters</td>
</tr>
<tr>
<td>Printer title</td>
<td>4,294,967,295 characters</td>
</tr>
<tr>
<td>Properties in a class</td>
<td>65,535</td>
</tr>
<tr>
<td>Queued resource lock</td>
<td>32,767 seconds</td>
</tr>
<tr>
<td>Range of Application::random method</td>
<td>32,767</td>
</tr>
<tr>
<td>Real variable</td>
<td>$1.7 \times 10^{-308}$ through $1.7 \times 10^{308}$</td>
</tr>
<tr>
<td>Rectangle width or height</td>
<td>32,767 units</td>
</tr>
<tr>
<td>References to a symbol (method name, variable, and so on)</td>
<td>65,535 in a JADE method</td>
</tr>
<tr>
<td>Rows in a table control or External Schema Wizard table</td>
<td>32,000</td>
</tr>
<tr>
<td>Schema entity name</td>
<td>100 characters</td>
</tr>
<tr>
<td>Server name</td>
<td>30 characters</td>
</tr>
<tr>
<td>Sheet of a table</td>
<td>16,000 columns and rows</td>
</tr>
<tr>
<td>String length</td>
<td>2,147,483,647 characters</td>
</tr>
<tr>
<td>String length in a user-defined StringArray</td>
<td>15,999 characters</td>
</tr>
<tr>
<td>StringUtf8 length in a user-defined StringUtf8Array</td>
<td>8,000 UTF8 characters</td>
</tr>
<tr>
<td>StringUtf8 length</td>
<td>2,147,483,647 characters</td>
</tr>
<tr>
<td>Sorted columns in a table</td>
<td>6 columns</td>
</tr>
<tr>
<td>Tab order</td>
<td>9 digits</td>
</tr>
<tr>
<td>Tab stops in the editor pane</td>
<td>64</td>
</tr>
<tr>
<td>Table control</td>
<td>63 sheets</td>
</tr>
<tr>
<td>Tag (form or control data value)</td>
<td>32,767 bytes</td>
</tr>
<tr>
<td>Timer interval</td>
<td>2,147,483,647 milliseconds</td>
</tr>
<tr>
<td>Translatable string name</td>
<td>100 characters</td>
</tr>
<tr>
<td>Translatable strings in a locale</td>
<td>65,535</td>
</tr>
<tr>
<td>User class number</td>
<td>999,999</td>
</tr>
<tr>
<td>User classes</td>
<td>997,952</td>
</tr>
<tr>
<td>User code</td>
<td>30 characters</td>
</tr>
<tr>
<td>User database files</td>
<td>65,286</td>
</tr>
<tr>
<td>User identification</td>
<td>29 characters (30, including the null terminator)</td>
</tr>
<tr>
<td>Web application pipes</td>
<td>62</td>
</tr>
<tr>
<td>Item</td>
<td>Maximum Limit</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>Width of form or control</td>
<td>32,767 pixels</td>
</tr>
<tr>
<td>Window object description</td>
<td>32,767 characters</td>
</tr>
</tbody>
</table>
# Appendix B

## Attribute Type Storage Sizes

The storage size for each attribute type is listed in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>1 byte</td>
</tr>
<tr>
<td>Byte</td>
<td>1 byte</td>
</tr>
<tr>
<td>Character</td>
<td>1 byte for ANSI, 2 bytes for Unicode</td>
</tr>
<tr>
<td>Embedded binaries</td>
<td>Schema-defined size + 2 bytes</td>
</tr>
<tr>
<td>Embedded strings</td>
<td>Schema-defined size + 3 bytes</td>
</tr>
<tr>
<td>Embedded StringUtf8</td>
<td>(Schema-defined size x 4) + 2 bytes</td>
</tr>
<tr>
<td>Date</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Decimal</td>
<td>3-12 bytes (depending on precision)</td>
</tr>
<tr>
<td>Integer</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Integer64</td>
<td>8 bytes</td>
</tr>
<tr>
<td>MemoryAddress</td>
<td>12 bytes</td>
</tr>
<tr>
<td>Point</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Non-embedded Binary Large Object (blob)</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Non-embedded String Large Object (slob)</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Non-embedded slob UTF8</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Real</td>
<td>8 bytes</td>
</tr>
<tr>
<td>Shared reference</td>
<td>12 bytes</td>
</tr>
<tr>
<td>Time</td>
<td>4 bytes</td>
</tr>
<tr>
<td>TimeStamp</td>
<td>8 bytes</td>
</tr>
<tr>
<td>TimeStampInterval</td>
<td>4 bytes</td>
</tr>
<tr>
<td>TimeStampOffset</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Unbounded binary and string attributes (referred to as blobs and slobs) are stored externally to normal object records.