Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related software documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007), Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.
Contents

Preface ..................................................................................................................................................... 7

1 Overview ............................................................................................................................................... 17
   Setting Up Your Environment ........................................................................................................ 17
   Starting and Testing a Message Broker ....................................................................................... 19
      ▼ To Start a Broker .................................................................................................................. 19
      ▼ To Test a Broker .................................................................................................................. 20
   Developing a Client Application ................................................................................................... 21
      ▼ To Produce Messages .............................................................................................................. 21
      ▼ To Consume Messages ........................................................................................................... 23
   Compiling and Running a Client Application ................................................................................. 26
      ▼ To Compile and Run the Hello World Message Application ............................................... 29
   Deploying a Client Application .................................................................................................... 30
   Example Application Code ............................................................................................................... 31

2 Using the Java API ............................................................................................................................... 33
   Messaging Domains ....................................................................................................................... 33
   Working With Connections .............................................................................................................. 34
      Obtaining a Connection Factory ............................................................................................... 35
      Using Connections ..................................................................................................................... 39
      Creating Secure Connections (SSL) .......................................................................................... 42
   Working With Destinations ............................................................................................................. 43
      Looking Up a Destination With JNDI ....................................................................................... 43
      Instantiating a Destination ........................................................................................................ 46
      Temporary Destinations ............................................................................................................. 47
   Working With Sessions .................................................................................................................... 47
   Acknowledgment Modes .................................................................................................................. 48
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transacted Sessions</td>
<td>50</td>
</tr>
<tr>
<td>Working With Messages</td>
<td>51</td>
</tr>
<tr>
<td>Message Structure</td>
<td>51</td>
</tr>
<tr>
<td>Composing Messages</td>
<td>56</td>
</tr>
<tr>
<td>Sending Messages</td>
<td>61</td>
</tr>
<tr>
<td>Receiving Messages</td>
<td>64</td>
</tr>
<tr>
<td>Processing Messages</td>
<td>71</td>
</tr>
<tr>
<td>3 Message Queue Clients: Design and Features</td>
<td>79</td>
</tr>
<tr>
<td>Client Design Considerations</td>
<td>79</td>
</tr>
<tr>
<td>Developing Portable Clients</td>
<td>80</td>
</tr>
<tr>
<td>Choosing Messaging Domains</td>
<td>80</td>
</tr>
<tr>
<td>Connections and Sessions</td>
<td>81</td>
</tr>
<tr>
<td>Producers and Consumers</td>
<td>82</td>
</tr>
<tr>
<td>Balancing Reliability and Performance</td>
<td>84</td>
</tr>
<tr>
<td>Managing Client Threads</td>
<td>84</td>
</tr>
<tr>
<td>JMS Threading Restrictions</td>
<td>85</td>
</tr>
<tr>
<td>Thread Allocation for Connections</td>
<td>85</td>
</tr>
<tr>
<td>Managing Memory and Resources</td>
<td>86</td>
</tr>
<tr>
<td>Managing Memory</td>
<td>86</td>
</tr>
<tr>
<td>Managing Message Size</td>
<td>86</td>
</tr>
<tr>
<td>Managing the Dead Message Queue</td>
<td>88</td>
</tr>
<tr>
<td>Managing Physical Destination Limits</td>
<td>92</td>
</tr>
<tr>
<td>Programming Issues for Message Consumers</td>
<td>92</td>
</tr>
<tr>
<td>Using the Client Runtime Ping Feature</td>
<td>92</td>
</tr>
<tr>
<td>Preventing Message Loss for Synchronous Consumers</td>
<td>93</td>
</tr>
<tr>
<td>Synchronous Consumption in Distributed Applications</td>
<td>93</td>
</tr>
<tr>
<td>Factors Affecting Performance</td>
<td>94</td>
</tr>
<tr>
<td>Delivery Mode (Persistent/Nonpersistent)</td>
<td>95</td>
</tr>
<tr>
<td>Use of Transactions</td>
<td>95</td>
</tr>
<tr>
<td>Acknowledgment Mode</td>
<td>95</td>
</tr>
<tr>
<td>Durable vs. Nondurable Subscriptions</td>
<td>96</td>
</tr>
<tr>
<td>Use of Selectors (Message Filtering)</td>
<td>97</td>
</tr>
<tr>
<td>Connection Event Notification</td>
<td>98</td>
</tr>
<tr>
<td>Connection Events</td>
<td>98</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Creating an Event Listener</td>
<td>99</td>
</tr>
<tr>
<td>Event Listener Examples</td>
<td>100</td>
</tr>
<tr>
<td>Client Connection Failover (Auto-Reconnect)</td>
<td>100</td>
</tr>
<tr>
<td>Enabling Auto-Reconnect</td>
<td>101</td>
</tr>
<tr>
<td>Auto-Reconnect Behaviors</td>
<td>104</td>
</tr>
<tr>
<td>Auto-Reconnect Limitations</td>
<td>105</td>
</tr>
<tr>
<td>Handling Exceptions When Failover Occurs</td>
<td>106</td>
</tr>
<tr>
<td>Custom Client Acknowledgment</td>
<td>135</td>
</tr>
<tr>
<td>Using Client Acknowledge Mode</td>
<td>135</td>
</tr>
<tr>
<td>Using No Acknowledge Mode</td>
<td>137</td>
</tr>
<tr>
<td>Schema Validation of XML Payload Messages</td>
<td>138</td>
</tr>
<tr>
<td>Communicating with Clients</td>
<td>139</td>
</tr>
<tr>
<td>Client Runtime Logging</td>
<td>139</td>
</tr>
<tr>
<td>Logging Name Spaces, Levels, and Activities</td>
<td>140</td>
</tr>
<tr>
<td>Using the JRE Logging Configuration File</td>
<td>142</td>
</tr>
<tr>
<td>Using a Logging Configuration File for a Specific Application</td>
<td>143</td>
</tr>
<tr>
<td>Setting the Logging Configuration Programmatically</td>
<td>143</td>
</tr>
<tr>
<td>4 Using the Metrics Monitoring API</td>
<td>145</td>
</tr>
<tr>
<td>Monitoring Overview</td>
<td>146</td>
</tr>
<tr>
<td>Administrative Tasks</td>
<td>147</td>
</tr>
<tr>
<td>Implementation Summary</td>
<td>147</td>
</tr>
<tr>
<td>Creating a Metrics-Monitoring Client</td>
<td>148</td>
</tr>
<tr>
<td>▼ To Monitor Broker Metrics</td>
<td>148</td>
</tr>
<tr>
<td>Format of Metrics Messages</td>
<td>149</td>
</tr>
<tr>
<td>Broker Metrics</td>
<td>149</td>
</tr>
<tr>
<td>Metrics Monitoring Client Code Examples</td>
<td>153</td>
</tr>
<tr>
<td>A Broker Metrics Example</td>
<td>153</td>
</tr>
<tr>
<td>5 Working with SOAP Messages</td>
<td>161</td>
</tr>
<tr>
<td>What is SOAP?</td>
<td>161</td>
</tr>
<tr>
<td>SOAP with Attachments API for Java</td>
<td>162</td>
</tr>
<tr>
<td>The SOAP Message</td>
<td>164</td>
</tr>
<tr>
<td>SOAP Packaging Models</td>
<td>165</td>
</tr>
<tr>
<td>SOAP Messaging in JAVA</td>
<td>167</td>
</tr>
</tbody>
</table>
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The SOAP Message Object</td>
<td>168</td>
</tr>
<tr>
<td>Destination, Message Factory, and Connection Objects</td>
<td>173</td>
</tr>
<tr>
<td>SOAP Messaging Models and Examples</td>
<td>174</td>
</tr>
<tr>
<td>SOAP Messaging Programming Models</td>
<td>174</td>
</tr>
<tr>
<td>Working with Attachments</td>
<td>175</td>
</tr>
<tr>
<td>Exception and Fault Handling</td>
<td>176</td>
</tr>
<tr>
<td>Writing a SOAP Client</td>
<td>177</td>
</tr>
<tr>
<td>Writing a SOAP Service</td>
<td>179</td>
</tr>
<tr>
<td>Integrating SOAP and Message Queue</td>
<td>185</td>
</tr>
<tr>
<td>Example 1: Deferring SOAP Processing</td>
<td>186</td>
</tr>
<tr>
<td>Example 2: Publishing SOAP Messages</td>
<td>189</td>
</tr>
<tr>
<td>Code Samples</td>
<td>190</td>
</tr>
<tr>
<td>6 Embedding a Message Queue Broker in a Java Client</td>
<td>197</td>
</tr>
<tr>
<td>Creating, Initializing and Starting an Embedded Broker</td>
<td>197</td>
</tr>
<tr>
<td>Creating a Broker Event Listener</td>
<td>198</td>
</tr>
<tr>
<td>Arguments to Specify When Initializing an Embedded Broker</td>
<td>199</td>
</tr>
<tr>
<td>Creating a Direct Connection to an Embedded Broker</td>
<td>200</td>
</tr>
<tr>
<td>Creating a TCP Connection to an Embedded Broker</td>
<td>200</td>
</tr>
<tr>
<td>Stopping and Shutting Down an Embedded Broker</td>
<td>200</td>
</tr>
<tr>
<td>Embedded Broker Example</td>
<td>200</td>
</tr>
<tr>
<td>A Warning Messages and Client Error Codes</td>
<td>203</td>
</tr>
<tr>
<td>Warning Messages and Error Codes</td>
<td>204</td>
</tr>
<tr>
<td>Index</td>
<td>217</td>
</tr>
</tbody>
</table>
Preface

This book provides information about concepts and procedures for developing Java messaging applications (Java clients) that work with Oracle GlassFish Message Queue.

This preface consists of the following sections:

- “Who Should Use This Book” on page 7
- “Before You Read This Book” on page 8
- “How This Book Is Organized” on page 8
- “Documentation Conventions” on page 8
- “Related Documentation” on page 11
- “Documentation, Support, and Training” on page 14
- “Searching Oracle Product Documentation” on page 14
- “Third-Party Web Site References” on page 15

Who Should Use This Book

This guide is meant principally for developers of Java applications that use Oracle GlassFish Message Queue.

These applications use the Java Message Service (JMS) Application Programming Interface (API), and possibly the SOAP with Attachments API for Java (SAAJ), to create, send, receive, and read messages. As such, these applications are JMS clients and/or SOAP client applications, respectively. The JMS and SAAJ specifications are open standards.

This book assumes that you are familiar with the JMS APIs and with JMS programming guidelines. Its purpose is to help you optimize your JMS client applications by making best use of the features and flexibility of a Message Queue messaging system.

This book assumes no familiarity, however, with SAAJ. This material is described in Chapter 5, “Working with SOAP Messages,” and assumes only basic knowledge of XML.
Before You Read This Book

You must read the Oracle GlassFish Message Queue 4.4.2 Technical Overview to become familiar with the Message Queue implementation of the Java Message Specification, with the components of the Message Queue service, and with the basic process of developing, deploying, and administering a Message Queue application.

How This Book Is Organized

This guide is designed to be read from beginning to end. The following table briefly describes the contents of each chapter.

TABLE P-1  Book Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1, “Overview”</td>
<td>A high-level overview of the Message Queue Java interface. It includes a tutorial that acquaints you with the Message Queue development environment using a simple example JMS client application.</td>
</tr>
<tr>
<td>Chapter 2, “Using the Java API”</td>
<td>Explains how to use the Message Queue Java API in your client application.</td>
</tr>
<tr>
<td>Chapter 3, “Message Queue Clients: Design and Features”</td>
<td>Describes architectural and configuration issues that depend upon Message Queue’s implementation of the Java Message Specification.</td>
</tr>
<tr>
<td>Chapter 4, “Using the Metrics Monitoring API”</td>
<td>Describes message-based monitoring, a customized solution to metrics gathering that allows metrics data to be accessed programmatically and then to be processed in whatever way suits the consuming client.</td>
</tr>
<tr>
<td>Chapter 5, “Working with SOAP Messages”</td>
<td>Explains how you send and receive SOAP messages with and without Message Queue support.</td>
</tr>
<tr>
<td>Appendix A, “Warning Messages and Client Error Codes”</td>
<td>Provides reference information for warning messages and error codes returned by the Message Queue client runtime when it raises a JMS exception.</td>
</tr>
</tbody>
</table>

Documentation Conventions

This section describes the following conventions used in Message Queue documentation:

- “Typographic Conventions” on page 9
- “Symbol Conventions” on page 9
- “Shell Prompt Conventions” on page 10
- “Directory Variable Conventions” on page 10
Typographic Conventions

The following table describes the typographic conventions that are used in this book.

<table>
<thead>
<tr>
<th>Typeface</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>AaBbCc123</td>
<td>The names of commands, files, and directories, and onscreen computer output</td>
<td>Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% you have mail</code>.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>What you type, contrasted with onscreen computer output</td>
<td><code>machine_name% su</code> Password: <code>aabbcc123</code></td>
</tr>
<tr>
<td>aabbcc123</td>
<td>Placeholder: replace with a real name or value</td>
<td>The command to remove a file is <code>rm filename</code>.</td>
</tr>
<tr>
<td>AaBbCc123</td>
<td>Book titles, new terms, and terms to be emphasized</td>
<td>Read Chapter 6 in the User’s Guide. A cache is a copy that is stored locally. Do not save the file. Note: Some emphasized items appear bold online.</td>
</tr>
</tbody>
</table>

Symbol Conventions

The following table explains symbols that might be used in this book.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>Contains optional arguments and command options.</td>
<td><code>ls [-l]</code></td>
<td>The <code>-l</code> option is not required.</td>
</tr>
<tr>
<td>{ }</td>
<td>Contains a set of choices for a required command option.</td>
<td>`-d {y</td>
<td>n}`</td>
</tr>
<tr>
<td>${ }</td>
<td>Indicates a variable reference.</td>
<td><code>${com.sun.javaRoot}</code></td>
<td>References the value of the <code>com.sun.javaRoot</code> variable.</td>
</tr>
<tr>
<td>-</td>
<td>Joins simultaneous multiple keystrokes.</td>
<td>Control-A</td>
<td>Press the Control key while you press the A key.</td>
</tr>
<tr>
<td>+</td>
<td>Joins consecutive multiple keystrokes.</td>
<td>Ctrl+A+N</td>
<td>Press the Control key, release it, and then press the subsequent keys.</td>
</tr>
</tbody>
</table>
Symbol Conventions (Continued)

Symbol | Description | Example | Meaning
--- | --- | --- | ---
→ | Indicates menu item selection in a graphical user interface. | File → New → Templates | From the File menu, choose New. From the New submenu, choose Templates.

Shell Prompt Conventions

The following table shows the conventions used in Message Queue documentation for the default UNIX system prompt and superuser prompt for the C shell, Bourne shell, Korn shell, and for the Windows operating system.

<table>
<thead>
<tr>
<th>Shell</th>
<th>Prompt</th>
</tr>
</thead>
<tbody>
<tr>
<td>C shell on UNIX, Linux, or AIX</td>
<td><code>machine-name%</code></td>
</tr>
<tr>
<td>C shell superuser on UNIX, Linux, or AIX</td>
<td><code>machine-name#</code></td>
</tr>
<tr>
<td>Bourne shell and Korn shell on UNIX, Linux, or AIX</td>
<td><code>$</code></td>
</tr>
<tr>
<td>Bourne shell and Korn shell superuser on UNIX, Linux, or AIX</td>
<td><code>#</code></td>
</tr>
<tr>
<td>Windows command line</td>
<td><code>C:\&gt;</code></td>
</tr>
</tbody>
</table>

Directory Variable Conventions

Message Queue documentation makes use of three directory variables; two of which represent environment variables needed by Message Queue. (How you set the environment variables varies from platform to platform.)

The following table describes the directory variables that might be found in this book and how they are used. When installed from the IPS (`pkg(5)`) image distribution, Message Queue is installed in a directory referred to as `mqInstallHome`, and some of the directory variables in Table P–5 reference this `mqInstallHome` directory.

Note – In this book, directory variables are shown without platform-specific environment variable notation or syntax (such as `$IMQ_HOME` on UNIX). Non-platform-specific path names use UNIX directory separator (`/`) notation.
Related Documentation

The information resources listed in this section provide further information about Message Queue in addition to that contained in this manual. The section covers the following resources:

- “Message Queue Documentation Set” on page 11
- “Java Message Service (JMS) Specification” on page 12
- “JavaDoc” on page 12
- "Example Client Applications” on page 13
- “Online Help” on page 14

Message Queue Documentation Set

The documents that constitute the Message Queue documentation set are listed in the following table in the order in which you might normally use them. These documents are available through the Oracle GlassFish Server documentation web site at

http://docs.sun.com/coll/1343.13
TABLE P–6  Message Queue Documentation Set

<table>
<thead>
<tr>
<th>Document</th>
<th>Audience</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle GlassFish Message Queue 4.4.2 Technical Overview</td>
<td>Developers and administrators</td>
<td>Describes Message Queue concepts, features, and components.</td>
</tr>
<tr>
<td>Oracle GlassFish Message Queue 4.4.2 Release Notes</td>
<td>Developers and administrators</td>
<td>Includes descriptions of new features, limitations, and known bugs, as well as technical notes.</td>
</tr>
<tr>
<td>Oracle GlassFish Message Queue 4.4.2 Administration Guide</td>
<td>Administrators, also recommended for developers</td>
<td>Provides background and information needed to perform administration tasks using Message Queue administration tools.</td>
</tr>
<tr>
<td>Oracle GlassFish Message Queue 4.4.2 Developer’s Guide for Java Clients</td>
<td>Developers</td>
<td>Provides a quick-start tutorial and programming information for developers of Java client programs using the Message Queue implementation of the JMS or SOAP/JAXM APIs.</td>
</tr>
<tr>
<td>Oracle GlassFish Message Queue 4.4.2 Developer’s Guide for C Clients</td>
<td>Developers</td>
<td>Provides programming and reference documentation for developers of C client programs using the Message Queue C implementation of the JMS API (C-API).</td>
</tr>
<tr>
<td>Oracle GlassFish Message Queue 4.4.2 Developer’s Guide for JMX Clients</td>
<td>Administrators</td>
<td>Provides programming and reference documentation for developers of JMX client programs using the Message Queue JMX API.</td>
</tr>
</tbody>
</table>

Java Message Service (JMS) Specification

The Message Queue message service conforms to the Java Message Service (JMS) application programming interface, described in the Java Message Service Specification. This document can be found at the URL

http://java.sun.com/products/jms/docs.html

JavaDoc

JMS and Message Queue API documentation in JavaDoc format is included in your Message Queue installation at the locations shown in Table P–7, depending on your installation method. This documentation can be viewed in any HTML browser. It includes standard JMS API documentation as well as Message Queue–specific APIs.
TABLE P–7  JavaDoc Locations

<table>
<thead>
<tr>
<th>Installation Method</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPS image</td>
<td>IMQ_HOME/javadoc/index.html¹</td>
</tr>
<tr>
<td>Solaris SVR4 packages</td>
<td>/usr/share/javadoc/imq/index.html</td>
</tr>
<tr>
<td>Linux RPM packages</td>
<td>/opt/sun/mq/javadoc/index.html</td>
</tr>
</tbody>
</table>

¹ IMQ_HOME is the Message Queue home directory.

Example Client Applications

Message Queue provides a number of example client applications to assist developers.

Example Java Client Applications

Example Java client applications are located in the following directories, depending on installation method. See the README files located in these directories and their subdirectories for descriptive information about the example applications.

<table>
<thead>
<tr>
<th>Installation Method</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPS image</td>
<td>IMQ_HOME/examples¹</td>
</tr>
<tr>
<td>Solaris SVR4 packages</td>
<td>/usr/demo/imq</td>
</tr>
<tr>
<td>Linux RPM packages</td>
<td>/opt/sun/mq/examples</td>
</tr>
</tbody>
</table>

¹ IMQ_HOME is the Message Queue home directory.

Example C Client Programs

Example C client applications are located in the following directories, depending on installation method. See the README files located in these directories and their subdirectories for descriptive information about the example applications.

<table>
<thead>
<tr>
<th>Installation Method</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPS image</td>
<td>IMQ_HOME/examples/C¹</td>
</tr>
<tr>
<td>Solaris SVR4 packages</td>
<td>/opt/SUNWimq/demo/C</td>
</tr>
<tr>
<td>Linux RPM packages</td>
<td>/opt/sun/mq/examples/C</td>
</tr>
</tbody>
</table>

¹ IMQ_HOME is the Message Queue home directory.
Example JMX Client Programs

Example Java Management Extensions (JMX) client applications are located in the following directories, depending on installation method. See the README files located in these directories and their subdirectories for descriptive information about the example applications.

<table>
<thead>
<tr>
<th>Installation Method</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPS image</td>
<td>IMQ_HOME/examples/jmx1</td>
</tr>
<tr>
<td>Solaris SVR4 packages</td>
<td>/opt/SUNWimq/demo/imq/jmx</td>
</tr>
<tr>
<td>Linux RPM packages</td>
<td>/opt/sun/mq/examples/jmx</td>
</tr>
</tbody>
</table>

1 IMQ_HOME is the Message Queue home directory.

Online Help

Online help is available for the Message Queue command line utilities; for details, see Chapter 16, Command Line Reference for details. The Message Queue graphical user interface (GUI) administration tool, the Administration Console, also includes a context-sensitive help facility; see the section "Administration Console Online Help" in Chapter 2, Quick-Start Tutorial.

Documentation, Support, and Training

The Oracle web site provides information about the following additional resources:

- Documentation (http://docs.sun.com/)
- Support (http://www.sun.com/support/)

Searching Oracle Product Documentation

Besides searching Oracle product documentation from the docs.sun.com web site, you can use a search engine by typing the following syntax in the search field:

search-term site:docs.sun.com

For example, to search for "broker," type the following:

broker site:docs.sun.com

To include other Oracle web sites in your search (for example, java.sun.com and developers.sun.com), use "sun .com" in place of "docs .sun .com" in the search field.
Third-Party Web Site References

Where relevant, this manual refers to third-party URLs that provide additional, related information.

Note – Oracle is not responsible for the availability of third-party Web sites mentioned in this manual. Oracle does not endorse and is not responsible or liable for any content, advertising, products, or other materials available on or through such sites or resources. Oracle will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with the use of or reliance on any such content, goods, or services available on or through such sites or resources.
Overview

This chapter provides an overall introduction to Oracle GlassFish Message Queue and a quick-start tutorial. It describes the procedures needed to create, compile, and run a simple example application. Before reading this chapter, you should be familiar with the concepts presented in the Oracle GlassFish Message Queue 4.4.2 Technical Overview.

The chapter covers the following topics:

- “Setting Up Your Environment” on page 17
- “Starting and Testing a Message Broker” on page 19
- “Developing a Client Application” on page 21
- “Compiling and Running a Client Application” on page 26
- “Deploying a Client Application” on page 30
- “Example Application Code” on page 31

The minimum Java Development Kit (JDK) level required to compile and run Message Queue clients is 1.2. For the purpose of this tutorial it is sufficient to run the Message Queue message broker in a default configuration. For instructions on configuring a message broker, see Chapter 4, “Configuring a Broker,” in Oracle GlassFish Message Queue 4.4.2 Administration Guide.

Setting Up Your Environment

The Message Queue files that need to be used in conjunction with Message Queue Java clients can be found in the lib directory in the installed location for Message Queue on your platform. Message Queue Java clients need to be able to use several .jar files found in the lib directory when these clients are compiled and run.

You need to set the CLASSPATH environment variable when compiling and running a JMS client.

The value of CLASSPATH depends on the following factors:

- The platform on which you compile or run
- The JDK version you are using
- Whether you are compiling or running a JMS application
- Whether your application uses the Simple Object Access Protocol (SOAP)
- Whether your application uses the SOAP/JMS transformer utilities

The directories where .jar files are stored depends on the installation method, as shown in the following table.

**TABLE 1-1 .jar File Locations**

<table>
<thead>
<tr>
<th>Installation Method</th>
<th>Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPS image</td>
<td>IMQ_HOME/lib/</td>
</tr>
<tr>
<td>Solaris™ SVR4 packages</td>
<td>/usr/share/lib/</td>
</tr>
<tr>
<td>Linux RPM packages</td>
<td>/opt/sun/mq/share/lib/</td>
</tr>
</tbody>
</table>

The table below lists the .jar files you need to compile and run different kinds of code.

**TABLE 1-2 .jar Files Needed in CLASSPATH**

<table>
<thead>
<tr>
<th>Type of Code</th>
<th>To Compile</th>
<th>To Run</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS client</td>
<td>jms.jar</td>
<td>jms.jar</td>
<td>See discussion of JNDI .jar files, following this table.</td>
</tr>
<tr>
<td></td>
<td>imq.jar</td>
<td>imq.jar</td>
<td></td>
</tr>
<tr>
<td></td>
<td>jndi.jar</td>
<td>jndi.jar</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Directory containing compiled Java application or '.',</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOAP Client</td>
<td>saaj-api.jar</td>
<td>saaj-api.jar</td>
<td>See Chapter 5, &quot;Working with SOAP Messages&quot;</td>
</tr>
<tr>
<td></td>
<td>activation.jar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOAP Servlet</td>
<td>jaxm-api.jar</td>
<td></td>
<td>GlassFish Server already includes these .jar files for SOAP servlet support.</td>
</tr>
<tr>
<td></td>
<td>saaj-api.jar</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>activation.jar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code using SOAP/JMS transformer utilities</td>
<td>imqxm.jar</td>
<td>imqxm.jar</td>
<td>Also add the appropriate .jar files listed in this table for the kind of code you are writing.</td>
</tr>
<tr>
<td></td>
<td>.jar files for JMS and SOAP clients</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A client application must be able to access the file jndi.jar even if the application does not use the Java Naming and Directory Interface (JNDI) directly to look up Message Queue administered objects. This is because JNDI is referenced by the Destination andConnectionFactory classes.
JNDI .jar files are bundled with JDK 1.4. Thus, if you are using this JDK, you do not have to add jndi.jar to your CLASSPATH setting. However, if you are using an earlier version of the JDK, you must include jndi.jar in your CLASSPATH.

If you are using JNDI to look up Message Queue administered objects, you must also include the following files in your CLASSPATH setting:

- If you are using the file-system service provider for JNDI (with any JDK version), you must include the file fscontext.jar.
- If you are using the Lightweight Directory Access Protocol (LDAP) context
  - with JDK 1.2 or 1.3, include the files ldabbp.jar, and fscontext.jar.ldap.jar,
  - with JDK 1.4, all files are already bundled with this JDK.

Starting and Testing a Message Broker

This tutorial assumes that you do not have a Message Queue message broker currently running. (If you run the broker as a UNIX startup process or Windows service, then it is already running and you can skip to “Developing a Client Application” on page 21.)

▼ To Start a Broker

1. In a terminal window, change to the directory containing Message Queue executables (see the table below).

<table>
<thead>
<tr>
<th>Installation Method</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPS Image</td>
<td>IMQ_HOME/bin</td>
</tr>
<tr>
<td>Solaris SVR4 packages</td>
<td>/usr/bin</td>
</tr>
<tr>
<td>Linux RPM packages</td>
<td>/opt/sun/mq/bin</td>
</tr>
</tbody>
</table>

2. Run the broker startup command (imqbrokerd) as follows:
   ```
   imqbrokerd -tty
   ```
   The -tty option causes all logged messages to be displayed to the terminal console (in addition to the log file). The broker will start and display a few messages before displaying the message
   ```
   imqbroker@host:7676 ready
   ```
   The broker is now ready and available for clients to use.
To Test a Broker

One simple way to check the broker startup is by using the Message Queue command utility (imqcmd) to display information about the broker:

1. In a separate terminal window, change to the directory containing Message Queue executables (see the table shown at the beginning of the section “To Start a Broker” on page 19).

2. Run imqcmd with the following arguments:

```
imqcmd query bkr -u admin
```

Supply the default password of admin when prompted to do so. The output displayed should be similar to that shown in the next example.

Example 1–1  Output From Testing a Broker

```
imqcmd query bkr -u admin

Querying the broker specified by:
-------------------------
Host    Primary Port
-------------------------
localhost 7676
Version 3.6
Instance Name imqbroker
Primary Port 7676
Current Number of Messages in System 0
Current Total Message Bytes in System 0
Max Number of Messages in System unlimited (-1)
Max Total Message Bytes in System unlimited (-1)
Max Message Size 70m
Auto Create Queues true
Auto Create Topics true
Auto Created Queue Max Number of Active Consumers 1
Auto Created Queue Max Number of Backup Consumers 0
Cluster Broker List (active)
Cluster Broker List (configured)
Cluster Master Broker
Cluster URL
```
Developing a Client Application

This section introduces the general procedures for interacting with the Message Queue API to produce and consume messages. The basic steps shown here are elaborated in greater detail in Chapter 2, “Using the Java API.” The procedures for producing and consuming messages have a number of steps in common, which need not be duplicated if the same client is performing both functions.

▼ To Produce Messages

1 Get a connection factory.

A Message Queue ConnectionFactory object encapsulates all of the needed configuration properties for creating connections to the Message Queue message service. You can obtain such an object either by direct instantiation.

   ConnectionFactory myFctry = new com.sun.messaging.ConnectionFactory();

or by looking up a predefined connection factory using the Java Naming and Directory Interface (JNDI). In the latter case, all of the connection factory’s properties will have been preconfigured to the appropriate values by your Message Queue administrator. If you instantiate the factory object yourself, you may need to configure some of its properties explicitly: for instance,

   myFctry.setProperty(ConnectionConfiguration.imqAddressList, "localhost:7676, broker2:5000, broker3:9999");
   myFctry.setProperty(ConnectionConfiguration.imqReconnectEnabled, "true");

See “Obtaining a Connection Factory” on page 35 for further discussion.

2 Create a connection.

A Connection object is an active connection to the Message Queue message service, created by the connection factory you obtained in “Developing a Client Application” on page 21:

   Connection myConnection = myFactory.createConnection();

See “Using Connections” on page 39 for further discussion.
3 **Create a session for communicating with the message service.**
A Session object represents a single-threaded context for producing and consuming messages. Every session exists within the context of a particular connection and is created by that connection's createSession method:

```java
Session mySession = myConnection.createSession(false,
                                               Session.AUTO_ACKNOWLEDGE);
```

The first (boolean) argument specifies whether the session is *transacted*. The second argument is the *acknowledgment mode*, such as AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, or DUPS_OK_ACKNOWLEDGE; these are defined as static constants in the JMS Session interface. See “Acknowledgment Modes” on page 48 and “Transacted Sessions” on page 50 for further discussion.

4 **Get a destination to which to send messages.**
A Destination object encapsulates provider-specific naming syntax and behavior for a message destination, which may be either a queue or a point-to-point publish/subscribe topic (see “Messaging Domains” on page 33). You can obtain such an object by direct instantiation:

```java
Destination myDest = new com.sun.messaging.Queue("myDest");
```

or by looking up a predefined destination using the JNDI API. See “Working With Destinations” on page 43 for further discussion.

5 **Create a message producer for sending messages to this destination.**
A MessageProducer object is created by a session and associated with a particular destination:

```java
MessageProducer myProducer = mySession.createProducer(myDest);
```

See “Sending Messages” on page 61 for further discussion.

6 **Create a message.**
A Session object provides methods for creating each of the six types of message defined by JMS: text, object, stream, map, bytes, and null messages. For instance, you can create a text message with the statement:

```java
TextMessage outMsg = mySession.createTextMessage();
```

See “Composing Messages” on page 56 for further discussion.

7 **Set the message’s content and properties.**
Each type of message has its own methods for specifying the contents of the message body. For instance, you can set the content of a text message with the statement:

```java
outMsg.setText("Hello, World!");
```

You can also use the property mechanism to define custom message properties of your own: for instance,

```java
outMsg.setStringProperty("MagicWord", "Shazam");
```
See “Working With Messages” on page 51 for further discussion.

8 **Send the message.**
The message producer’s send method sends a message to the destination with which the producer is associated:
myProducer.send(outMsg);
See “Sending Messages” on page 61 for further discussion.

9 **Close the session.**
When there are no more messages to send, you should close the session
mySession.close();
allowing Message Queue to free any resources it may have associated with the session. See “Working With Sessions” on page 47 for further discussion.

10 **Close the connection.**
When all sessions associated with a connection have been closed, you should close the connection by calling its close method:
myConnection.close();
See “Using Connections” on page 39 for further discussion.

▼ **To Consume Messages**

1 **Get a connection factory.**
A Message QueueConnectionFactory object encapsulates all of the needed configuration properties for creating connections to the Message Queue message service. You can obtain such an object either by direct instantiation
ConnectionFactory myFctry = new com.sun.messaging.ConnectionFactory();
or by looking up a predefined connection factory using the Java Naming and Directory Interface (JNDI). In the latter case, all of the connection factory’s properties will have been preconfigured to the appropriate values by your Message Queue administrator. If you instantiate the factory object yourself, you may need to configure some of its properties explicitly: for instance,

myFctry.setProperty(ConnectionConfiguration.imqAddressList,
    "localhost:7676, broker2:5000, broker3:9999");
myFctry.setProperty(ConnectionConfiguration.imqReconnectEnabled,"true");
See “Obtaining a Connection Factory” on page 35 for further discussion.
2  **Create a connection.**

   A Connection object is an active connection to the Message Queue message service, created by the connection factory you obtained in “Developing a Client Application” on page 21:

   ```java
   Connection myConnection = myFactory.createConnection();
   ```

   See “Using Connections” on page 39 for further discussion.

3  **Create a session for communicating with the message service.**

   A Session object represents a single-threaded context for producing and consuming messages. Every session exists within the context of a particular connection and is created by that connection's `createSession` method:

   ```java
   Session mySession = myConnection.createSession(false,
                                      Session.AUTO_ACKNOWLEDGE);
   ```

   The first (boolean) argument specifies whether the session is transacted. The second argument is the acknowledgment mode, such as AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, or DUPS_OK_ACKNOWLEDGE; these are defined as static constants in the JMS Session interface. See “Acknowledgment Modes” on page 48 and “Transacted Sessions” on page 50 for further discussion.

4  **Get a destination from which to receive messages.**

   A Destination object encapsulates provider-specific naming syntax and behavior for a message destination, which may be either a point-to-point queue or a publish/subscribe topic (see “Messaging Domains” on page 33). You can obtain such an object by direct instantiation

   ```java
   Destination myDest = new com.sun.messaging.Queue("myDest");
   ```

   or by looking up a predefined destination using the JNDI API. See “Working With Destinations” on page 43 for further discussion.

5  **Create a message consumer for receiving messages from this destination.**

   A MessageConsumer object is created by a session and associated with a particular destination:

   ```java
   MessageConsumer myConsumer = mySession.createConsumer(myDest);
   ```

   See “Receiving Messages” on page 64 for further discussion.

6  **Start the connection.**

   In order for a connection's message consumers to begin receiving messages, you must start the connection by calling its `start` method:

   ```java
   myConnection.start();
   ```

   See “Using Connections” on page 39 for further discussion.
7 Receive a message.
The message consumer's receive method requests a message from the destination with which
the consumer is associated:
Message inMsg = myConsumer.receive();
This method is used for synchronous consumption of messages. You can also configure a
message consumer to consume messages asynchronously, by creating a message listener and
associating it with the consumer. See "Receiving Messages" on page 64 for further discussion.

8 Retrieve the message's content and properties.
Each type of message has its own methods for extracting the contents of the message body. For
instance, you can retrieve the content of a text message with the statements
TextMessage txtMsg = (TextMessage) inMsg;
String msgText = txtMsg.getText();
In addition, you may need to retrieve some of the message's header fields: for instance,
msgPriority = inMsg.getJMSPriority();
You can also use message methods to retrieve custom message properties of your own: for
instance,
magicWord = inMsg.getStringProperty("MagicWord");
See “Processing Messages” on page 71 for further discussion.

9 Close the session.
When there are no more messages to consume, you should close the session
mySession.close();
allowing Message Queue to free any resources it may have associated with the session. See
"Working With Sessions" on page 47 for further discussion.

10 Close the connection.
When all sessions associated with a connection have been closed, you should close the
connection by calling its close method:
myConnection.close();
See "Using Connections" on page 39 for further discussion.
Compiling and Running a Client Application

This section leads you through the steps needed to compile and run a simple example client application, *HelloWorldMessage*, that sends a message to a destination and then retrieves the same message from the destination. The code shown in Example 1–2 is adapted and simplified from an example program provided with the Message Queue installation: error checking and status reporting have been removed for the sake of conceptual clarity. You can find the complete original program in the `helloworld` directory in the following locations.

- Solaris: `/usr/demo/imq/`
- Linux: `opt/sun/mq/examples`
- Windows: `IMQ_HOME/demo`

Example 1–2  Simple Message Queue Client Application

```java
// Import the JMS and JNDI API classes
import javax.jms.*;
import javax.naming.*;
import java.util.Hashtable;

public class HelloWorldMessage
{
   /**
    * Main method
    *
    * Parameter args not used
    *
    */
   public static void main (String[] args)
   {
      try
      {
         // Get a connection factory.
         //
         // Create the environment for constructing the initial JNDI
         // naming context.
         Hashtable env = new Hashtable();

         // Store the environment attributes that tell JNDI which
         // initial context
         // factory to use and where to find the provider.
         // (On Unix, use provider URL "file:///imq_admin_objects"
```

```
// instead of "file:///C:/imq_admin_objects".)

env.put(Context.INITIAL_CONTEXT_FACTORY, 
    "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL,"file:///C:/imq_admin_objects");

// Create the initial context.
Context ctx = new InitialContext(env);

// Look up connection factory object in the JNDI object store.
String CF_LOOKUP_NAME = "MyConnectionFactory";
ConnectionFactory myFactory = 
    (ConnectionFactory) ctx.lookup(CF_LOOKUP_NAME);

// Create a connection.
Connection myConnection = myFactory.createConnection();

// Create a session.
Session mySession = myConnection.createSession(false, 
    Session.AUTO_ACKNOWLEDGE);

// Look up the destination object in the JNDI object store.
String DEST_LOOKUP_NAME = "MyDest";
Destination myDest = (Destination) ctx.lookup 
    (DEST_LOOKUP_NAME);

// Create a message producer.
MessageProducer myProducer = mySession.createProducer(myDest);

// Create a message consumer.
MessageConsumer myConsumer = mySession.createConsumer(myDest);
// Create a message.

TextMessage outMsg = mySession.createTextMessage("Hello, World!");

// Send the message to the destination.

System.out.println("Sending message: "+ outMsg.getText());
myProducer.send(outMsg);

// Start the connection.

myConnection.start();

// Receive a message from the destination.

Message inMsg = myConsumer.receive();

// Retrieve the contents of the message.

if (inMsg instanceof TextMessage)
{
    TextMessage txtMsg = (TextMessage) inMsg;
    System.out.println("Received message: "+
                        txtMsg.getText());
}

// Close the session and the connection.

mySession.close();
myConnection.close();

}

} catch (Exception jmse)
{
    System.out.println("Exception occurred: "+ jmse.toString());
    jmse.printStackTrace();
}
To compile and run Java clients in a Message Queue environment, it is recommended that you use the Java 2 SDK, Standard Edition, version 1.4 or later. You can download the recommended SDK from the following location:

http://java.sun.com/j2se/1.5

Be sure to set your CLASSPATH environment variable correctly, as described in "Setting Up Your Environment" on page 17, before attempting to compile or run a client application.

Note – If you are using JDK 1.5, you will get compiler errors if you use the unqualified JMS Queue class along with the following import statement.

import java.util.*

This is because the packages java.util and javax.jms both contain a class named Queue. To avoid the compilation errors, you must eliminate the ambiguity by either fully qualifying references to the JMS Queue class as javax.jms.Queue or correcting your import statements to refer to specific individual java.util classes.

The following steps for compiling and running the HelloWorldMessage application are furnished strictly as an example. The program is shipped precompiled; you do not actually need to compile it yourself (unless, of course, you modify its source code).

▼ To Compile and Run the HelloWorldMessage Application

1 Make the directory containing the application your current directory.

   The Message Queue example applications directory on Solaris is not writable by users, so copy the HelloWorldMessage application to a writable directory and make that directory your current directory.

2 Compile the HelloWorldMessage application:

   javac HelloWorldMessage.java

   This creates the file HelloWorldMessage.class in your current directory.
3 Run the **HelloWorldMessage** application:

```java
java HelloWorldMessage
```

The program should display the following output:

```
Sending Message: Hello, World!
Received Message: Hello, World!
```

### Deploying a Client Application

When you are ready to deploy your client application, you should make sure your Message Queue administrator knows your application’s needs. The checklist shown below summarizes the information required; consult with your administrator for specific details. In some cases, it may be useful to provide a range of values rather than a specific value. See Chapter 11, "Managing Administered Objects," in *Oracle GlassFish Message Queue 4.4.2 Administration Guide* for details on configuration and on attribute names and default values for administered objects.

- **Administered Objects**
  - Connection Factories
    - Type
    - JNDI lookup name
    - Other attributes
  - Destinations
    - Type (queue or topic)
    - JNDI lookup name
    - Physical destination name
- **Physical Destinations**
  - Type
  - Name
  - Attributes
  - Maximum number of messages expected
  - Maximum size of messages expected
  - Maximum message bytes expected
- **Broker or Broker Cluster**
  - Name
  - Port
  - Properties
- **Dead Message Queue**
  - Place dead messages on dead message queue?
  - Log placement of messages on dead message queue?
Example Application Code

The Message Queue installation includes example programs illustrating both JMS and JAXM messaging (see Chapter 5, “Working with SOAP Messages”). They are located in the following directories:

- **Installation from IPS image:** IMQ_HOME/examples
- **Installation from Solaris SVR4 packages:** /usr/demo/imq
- **Installation from Linux RPM packages:** /opt/sun/mq/examples

Each directory (except the JMS directory) contains a README file describing the source files included in that directory. The table below lists the directories of interest to Message Queue Java clients.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>helloworld</td>
<td>Sample programs showing how to create and deploy a JMS client in Message Queue, including the steps required to create administered objects and to look up such objects with JNDI from within client code</td>
</tr>
<tr>
<td>jms</td>
<td>Sample programs demonstrating the use of the JMS API with Message Queue</td>
</tr>
<tr>
<td>jaxm</td>
<td>Sample programs demonstrating the use of SOAP messages in conjunction with JMS in Message Queue</td>
</tr>
<tr>
<td>applications</td>
<td>Four subdirectories containing source code for the following:</td>
</tr>
<tr>
<td></td>
<td>- A GUI application using the JMS API to implement a simple chat application</td>
</tr>
<tr>
<td></td>
<td>- A GUI application using the Message Queue JMS monitoring API to obtain a list of queues from a Message Queue broker and browse their contents with a JMS queue browser</td>
</tr>
<tr>
<td></td>
<td>- The Message Queue Ping demo program</td>
</tr>
<tr>
<td></td>
<td>- The Message Queue Applet demo program</td>
</tr>
<tr>
<td>monitoring</td>
<td>Sample programs demonstrating the use of the JMS API to monitor a message broker</td>
</tr>
<tr>
<td>jdbc</td>
<td>Examples for plugging in a PointBase and an Oracle database</td>
</tr>
<tr>
<td>imqobjmgr</td>
<td>Examples of imqobjmgr command files</td>
</tr>
</tbody>
</table>
Using the Java API

This chapter describes how to use the classes and methods of the Message Queue Java application programming interface (API) to accomplish specific tasks, and provides brief code samples to illustrate some of these tasks. (For clarity, the code samples shown in the chapter omit an exception check.) The topics covered include the following:

- “Messaging Domains” on page 33
- "Working With Connections” on page 34
- “Working With Destinations” on page 43
- “Working With Sessions” on page 47
- “Working With Messages” on page 51

This chapter does not provide exhaustive information about each class and method. For detailed reference information, see the JavaDoc documentation for each individual class. For information on the practical design of Message Queue Java programs, see Chapter 3, “Message Queue Clients: Design and Features”

Messaging Domains

The Java Message Service (JMS) specification, which Message Queue implements, supports two commonly used models of interaction between message clients and message brokers, sometimes known as messaging domains:

- In the point-to-point (or PTP) messaging model, each message is delivered from a message producer to a single message consumer. The producer delivers the message to a queue, from which it is later delivered to one of the consumers registered for the queue. Any number of producers and consumers can interact with the same queue, but each message is guaranteed to be delivered to (and be successfully consumed by) exactly one consumer and no more. If no consumers are registered for a queue, it holds the messages it receives and eventually delivers them when a consumer registers.
In the publish/subscribe (or pub/sub) model, a single message can be delivered from a producer to any number of consumers. The producer publishes the message to a topic, from which it is then delivered to all active consumers that have subscribed to the topic. Any number of producers can publish messages to a given topic, and each message can be delivered to any number of subscribed consumers. The model also supports the notion of durable subscriptions, in which a consumer registered with a topic need not be active at the time a message is published; when the consumer subsequently becomes active, it will receive the message. If no active consumers are registered for a topic, the topic does not hold the messages it receives unless it has inactive consumers with durable subscriptions.

JMS applications are free to use either of these messaging models, or even to mix them both within the same application. Historically, the JMS API provided a separate set of domain-specific object classes for each model. While these domain-specific interfaces continue to be supported for legacy purposes, client programmers are now encouraged to use the newer unified domain interface, which supports both models indiscriminately. For this reason, the discussions and code examples in this manual focus exclusively on the unified interfaces wherever possible. Table 2–1 shows the API classes for all three domains.

<table>
<thead>
<tr>
<th>Unified Domain</th>
<th>Point-to-Point Domain</th>
<th>Publish/Subscribe Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination</td>
<td>Queue</td>
<td>Topic</td>
</tr>
<tr>
<td>ConnectionFactory</td>
<td>QueueConnectionFactory</td>
<td>TopicConnectionFactory</td>
</tr>
<tr>
<td>Connection</td>
<td>QueueConnection</td>
<td>TopicConnection</td>
</tr>
<tr>
<td>Session</td>
<td>QueueSession</td>
<td>TopicSession</td>
</tr>
<tr>
<td>MessageProducer</td>
<td>QueueSender</td>
<td>TopicPublisher</td>
</tr>
<tr>
<td>MessageConsumer</td>
<td>QueueReceiver</td>
<td>TopicSubscriber</td>
</tr>
</tbody>
</table>

Working With Connections

All messaging occurs within the context of a connection. Connections are created using a connection factory encapsulating all of the needed configuration properties for connecting to a particular JMS provider. A connection’s configuration properties are completely determined by the connection factory, and cannot be changed once the connection has been created. Thus the only way to control the properties of a connection is by setting those of the connection factory you use to create it.
Obtaining a Connection Factory

Typically, a connection factory is created for you by a Message Queue administrator and preconfigured, using the administration tools described in the Chapter 1, “Administrative Tasks and Tools,” in Oracle GlassFish Message Queue 4.4.2 Administration Guide with whatever property settings are appropriate for connecting to particular JMS provider. The factory is then placed in a publicly available administered object store, where you can access it by name using the Java Naming and Directory Interface (JNDI) API. This arrangement has several benefits:

- It allows the administrator to control the properties of client connections to the provider, ensuring that they are properly configured.
- It enables the administrator to tune performance and improve throughput by adjusting configuration settings even after an application has been deployed.
- By relying on the predefined connection factory to handle the configuration details, it helps keep client code provider-independent and thus more easily portable from one JMS provider to another.

Sometimes, however, it may be more convenient to dispense with JNDI lookup and simply create your own connection factory by direct instantiation. Although hard-coding configuration values for a particular JMS provider directly into your application code sacrifices flexibility and provider-independence, this approach might make sense in some circumstances: for example, in the early stages of application development and debugging, or in applications where reconfigurability and portability to other providers are not important concerns.

The following sections describe these two approaches to obtaining a connection factory: by JNDI lookup or direct instantiation.

Looking Up a Connection Factory With JNDI

Example 2-1 shows how to look up a connection factory object in the JNDI object store. The code example is explained in the procedure that follows.

Note – If a Message Queue client is a J2EE component, JNDI resources are provided by the J2EE container. In such cases, JNDI lookup code may differ from that shown here; see your J2EE provider documentation for details.

EXAMPLE 2-1  Looking Up a Connection Factory

```java
// Create the environment for constructing the initial JNDI
// naming context.

Hashtable env = new Hashtable();
```
EXAMPLE 2–1  Looking Up a Connection Factory  (Continued)

// Store the environment attributes that tell JNDI which initial context
// factory to use and where to find the provider. //

    env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.fscontext.RefFSContextFactory");
    env.put(Context.PROVIDER_URL, "file:///C:/imq_admin_objects");

// Create the initial context.

    Context ctx = new InitialContext(env);

// Look up the connection factory object in the JNDI object store.

    String CF_LOOKUP_NAME = "MyConnectionFactory";
    ConnectionFactory myFactory = (ConnectionFactory) ctx.lookup
        (CF_LOOKUP_NAME);


▼ To Look Up a Connection Factory With JNDI

1 Create the environment for constructing the initial JNDI naming context.

How you create the initial context depends on whether you are using a file-system object store
or a Lightweight Directory Access Protocol (LDAP) server for your Message Queue
administered objects. The code shown here assumes a file-system store; for information about
the corresponding LDAP object store attributes, see "Using an LDAP User Repository" in
Oracle GlassFish Message Queue 4.4.2 Administration Guide

The constructor for the initial context accepts an environment parameter, a hash table whose
entries specify the attributes for creating the context:

    Hashtable env = new Hashtable();

You can also set an environment by specifying system properties on the command line, rather
than programmatically. For instructions, see the README file in the JMS example applications
directory.

2 Store the environment attributes that tell JNDI which initial context factory to use and where to
find the JMS provider.

The names of these attributes are defined as static constants in class Context:

    env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.fscontext.RefFSContextFactory");
    env.put(Context.PROVIDER_URL, "file:///C:/imq_admin_objects");
Note – The directory represented by `C:/imq_admin_objects` must already exist; if necessary, you must create the directory before referencing it in your code.

3 Create the initial context.

```java
Context ctx = new InitialContext(env);
```

If you use system properties to set the environment, omit the environment parameter when creating the context:

```java
Context ctx = new InitialContext();
```

4 Look up the connection factory object in the administered object store and typecast it to the appropriate class:

```java
String CF_LOOKUP_NAME = "MyConnectionFactory";
ConnectionFactory
    myFactory = (ConnectionFactory) ctx.lookup(CF_LOOKUP_NAME);
```

The lookup name you use, `CF_LOOKUP_NAME`, must match the name used when the object was stored.

You can now proceed to use the connection factory to create connections to the message broker, as described under “Using Connections” on page 39.

Overriding Configuration Settings

It is recommended that you use a connection factory just as you receive it from a JNDI lookup, with the property settings originally configured by your Message Queue administrator. However, there may be times when you need to override the preconfigured properties with different values of your own. You can do this from within your application code by calling the connection factory's `setProperty` method. This method (inherited from the superclass `AdministeredObject`) takes two string arguments giving the name and value of the property to be set. The property names for the first argument are defined as static constants in the Message Queue class `ConnectionConfiguration`: for instance, the statement

```java
myFactory.setProperty(ConnectionConfiguration.imqDefaultPassword, "mellon");
```

sets the default password for establishing broker connections. See “Connection Factory Attributes” in Oracle GlassFish Message Queue 4.4.2 Administration Guide for complete information on the available connection factory configuration attributes.

It is also possible to override connection factory properties from the command line, by using the `-D` option to set their values when starting your client application. For example, the command line

```bash
java -DmqDefaultPassword=mellon MyMQClient
```
starts an application named MyMQClient with the same default password as in the preceding example. Setting a property value this way overrides any other value specified for it, whether preconfigured in the JNDI object store or set programmatically with the setProperty method.

**Note** – A Message Queue administrator can prevent a connection factory’s properties from being overridden by specifying that the object be read-only when placing it in the object store. The properties of such a factory cannot be changed in any way, whether with the -D option from the command line or using the setProperty method from within your client application’s code. Any attempt to override the factory’s property values will simply be ignored.

### Instantiating a Connection Factory

**Example 2–2** shows how to create a connection factory object by direct instantiation and configure its properties.

**EXAMPLE 2–2**  Instantiating a Connection Factory

```java
// Instantiate the connection factory object.
com.sun.messaging.ConnectionFactory
    myFactory = new com.sun.messaging.ConnectionFactory();

// Set the connection factory's configuration properties.
myFactory.setProperty(ConnectionConfiguration.imqAddressList,
                      "localhost:7676,broker2:5000,broker3:9999");
```

The following procedure explains each program statement in the previous code sample.

**▼ To Instantiate and Configure a Connection Factory**

1 **Instantiate the connection factory object.**

   The name ConnectionFactory is defined both as a JMS interface (in package javax.jms) and as a Message Queue class (in com.sun.messaging) that implements that interface. Since only a class can be instantiated, you must use the constructor defined in com.sun.messaging to create your connection factory object. Note, however, that you cannot import the name from both
packages without causing a compilation error. Hence, if you have imported the entire package javax.jms.*, you must qualify the constructor with the full package name when instantiating the object:

```java
com.sun.messaging.ConnectionFactory
    myFactory = new com.sun.messaging.ConnectionFactory();
```

Notice that the type declaration for the variable myFactory, to which the instantiated connection factory is assigned, is also qualified with the full package name. This is because the `setProperty` method, used in "Instantiating a Connection Factory" on page 38, belongs to the `ConnectionFactory` class defined in the package `com.sun.messaging`, rather than to the `ConnectionFactory` interface defined in `javax.jms`. Thus in order for the compiler to recognize this method, `myFactory` must be typed explicitly as `com.sun.messaging.ConnectionFactory` rather than simply `ConnectionFactory` (which would resolve to `javax.jms.ConnectionFactory` after importing `javax.jms.*`).

2 Set the connection factory's configuration properties.

The most important configuration property is `imqAddressList`, which specifies the host names and port numbers of the message brokers to which the factory creates connections. By default, the factory returned by the `ConnectionFactory` constructor in "Instantiating a Connection Factory" on page 38 is configured to create connections to a broker on host localhost at port number 7676. If necessary, you can use the `setProperty` method, described in the preceding section, to change that setting:

```java
myFactory.setProperty(ConnectionConfiguration.imqAddressList,
    "localhost:7676,broker2:5000,broker3:9999");
```

When specifying the host name portion of a broker, you can use a literal IPv4 or IPv6 address instead of a host name. If you use a literal IPv6 address, its format must conform to RFC2732, Format for Literal IPv6 Addresses in URL’s.

Of course, you can also set any other configuration properties your application may require. See "Connection Factory Attributes" in Oracle GlassFish Message Queue 4.4.2 Administration Guide for a list of the available connection factory attributes.

You can now proceed to use the connection factory to create connections to the message service, as described in the next section.

Using Connections

Once you have obtained a connection factory, you can use it to create a connection to the message service. The factory's `createConnection` method takes a user name and password as arguments:

```java
Connection
    myConnection = myFactory.createConnection("mithrandir", "mellon");
```
Before granting the connection, Message Queue authenticates the user name and password by looking them up in its user repository. As a convenience for developers who do not wish to go to the trouble of populating a user repository during application development and testing, there is also a parameterless form of the `createConnection` method:

```java
Connection myConnection = myFactory.createConnection();
```

This creates a connection configured for the default user identity, with both user name and password set to `guest`.

This unified-domain `createConnection` method is part of the generic JMS `ConnectionFactory` interface, defined in package `javax.jms`; the Message Queue version in `com.sun.messaging` adds corresponding methods `createQueueConnection` and `createTopicConnection` for use specifically with the point-to-point and publish/subscribe domains.

The following table shows the methods defined in the `Connection` interface.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createSession</code></td>
<td>Create session</td>
</tr>
<tr>
<td><code>setClientID</code></td>
<td>Set client identifier</td>
</tr>
<tr>
<td><code>getClientID</code></td>
<td>Get client identifier</td>
</tr>
<tr>
<td><code>setEventListener</code></td>
<td>Set event listener for connection events</td>
</tr>
<tr>
<td><code>setExceptionListener</code></td>
<td>Set exception listener</td>
</tr>
<tr>
<td><code>getExceptionListener</code></td>
<td>Get exception listener</td>
</tr>
<tr>
<td><code>getMetaData</code></td>
<td>Get metadata for connection</td>
</tr>
<tr>
<td><code>createConnectionConsumer</code></td>
<td>Create connection consumer</td>
</tr>
<tr>
<td><code>createDurableConnectionConsumer</code></td>
<td>Create durable connection consumer</td>
</tr>
<tr>
<td><code>start</code></td>
<td>Start incoming message delivery</td>
</tr>
<tr>
<td><code>stop</code></td>
<td>Stop incoming message delivery</td>
</tr>
<tr>
<td><code>close</code></td>
<td>Close connection</td>
</tr>
</tbody>
</table>

The main purpose of a connection is to create sessions for exchanging messages with the message service:

```java
myConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```
The first argument to createSession is a boolean indicating whether the session is transacted; the second specifies its acknowledgment mode. Possible values for this second argument are AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, and DUPS_OK_ACKNOWLEDGE, all defined as static constants in the standard JMS Session interface, java.jms.Session; the extended Message Queue version of the interface, com.sun.messaging.jms.Session, adds another such constant, NO_ACKNOWLEDGE. See “Acknowledgment Modes” on page 48 and “Transacted Sessions” on page 50 for further discussion.

If your client application will be using the publish/subscribe domain to create durable topic subscriptions, it must have a client identifier to identify itself to the message service. In general, the most convenient arrangement is to configure the client runtime to provide a unique client identifier automatically for each client. However, the Connection interface also provides a method, setClientID, for setting a client identifier explicitly, and a corresponding getClientID method for retrieving its value. See "Assigning Client Identifiers" on page 82 and "Client Identifier" in Oracle GlassFish Message Queue 4.4.2 Administration Guide for more information.

You should also use the setExceptionListener method to register an exception listener for the connection. This is an object implementing the JMS ExceptionListener interface, which consists of the single method onException:

```
void onException (JMSException exception)
```

In the event of a problem with the connection, the message broker will call this method, passing an exception object identifying the nature of the problem.

A connection's getMetaData method returns a ConnectionMetaData object, which in turn provides methods for obtaining various items of information about the connection, such as its JMS version and the name and version of the JMS provider.

The createConnectionConsumer and createDurableConnectionConsumer methods (as well as the session methods setMessageListener and getMessageListener, listed in Table 2–3) are used for concurrent message consumption; see the Java Message Service Specification for more information.

In order to receive incoming messages, you must start the connection by calling its start method:

```
myConnection.start();
```

It is important not to do this until after you have created any message consumers you will be using to receive messages on the connection. Starting the connection before creating the consumers risks missing some incoming messages before the consumers are ready to receive them. It is not necessary to start the connection in order to send outgoing messages.

If for any reason you need to suspend the flow of incoming messages, you can do so by calling the connection’s stop method:
myConnection.stop();

To resume delivery of incoming messages, call the start method again.

Finally, when you are through with a connection, you should close it to release any resources associated with it:

myConnection.close();

This automatically closes all sessions, message producers, and message consumers associated with the connection and deletes any temporary destinations. All pending message receives are terminated and any transactions in progress are rolled back. Closing a connection does not force an acknowledgment of client-acknowledged sessions.

Creating Secure Connections (SSL)

A connection service that is based on the Transport Layer Security (TLS/SSL) standard is used to authenticate and encrypt messages sent between the client and the broker. This section describes what the client needs to do to use TLS/SSL connections. A user can also establish a secure connection by way of an HTTPS tunnel servlet. For information on setting up secure connections over HTTP, see Appendix C, "HTTP/HTTPS Support," in Oracle GlassFish Message Queue 4.4.2 Administration Guide.

Some of the work needed to set up a TLS/SSL connection is done by an administrator. This section summarizes these steps. For complete information about the administrative work required, please see "Message Encryption" in Oracle GlassFish Message Queue 4.4.2 Administration Guide.

To set up a secure connection service, you must do the following.

1. Generate a self-signed or signed certificate for the broker (administrator).
2. Enable the ssl.jms connection service in the broker (administrator).
3. Start the broker (administrator).
4. Configure and run the client as explained below.

To configure a client to use a TLS/SSL connection you must do the following.

1. If your client is not using J2SDK 1.4 (which has JSSE and JNDI support built in), make sure the client has the following files in its class path: jsse.jar, jnet.jar, jcert.jar, jndi.jar.
2. Make sure the client has the following Message Queue files in its class path: imq.jar, jms.jar.
3. If the client is not willing to trust the broker's self-signed certificate, set the imqSSLIsHostTrusted attribute to false for the connection factory from which you get the TLS/SSL connection.
4. Connect to the broker’s ssljms service. There are two ways to do this. The first is to specify the service name ssljms in the address for the broker when you provide a value for the imqAddressList attribute of the connection factory from which you obtain the connection. When you run the client, it will be connected to the broker by a TLS/SSL connection. The second is to specify the following directive when you run the command that starts the client.

   java -DimqConnectionType=TLS clientAppName

Working With Destinations

All Message Queue messages travel from a message producer to a message consumer by way of a destination on a message broker. Message delivery is thus a two-stage process: the message is first delivered from the producer to the destination and later from the destination to the consumer. Physical destinations on the broker are created administratively by a Message Queue administrator, using the administration tools described in “Configuring and Managing Physical Destinations” in Oracle GlassFish Message Queue 4.4.2 Administration Guide. The broker provides routing and delivery services for messages sent to such a destination.

As described earlier under "Messaging Domains" on page 33, Message Queue supports two types of destination, depending on the messaging domain being used:

- Queues (point-to-point domain)
- Topics (publish/subscribe domain)

These two types of destination are represented by the Message Queue classes Queue and Topic, respectively. These, in turn, are both subclasses of the generic class Destination, part of the unified messaging domain that subsumes both the point-to-point and publish-subscribe domains. A client program that uses the Destination superclass can thus handle both queue and topic destinations indiscriminately.

Looking Up a Destination With JNDI

Because JMS providers differ in their destination addressing conventions, Message Queue does not define a standard address syntax for obtaining access to a destination. Rather, the destination is typically placed in a publicly available administered object store by a Message Queue administrator and accessed by the client using a JNDI lookup in a manner similar to that described earlier for connection factories (see “Looking Up a Connection Factory With JNDI” on page 35).

Example 2–3 shows how to look up a destination object in the JNDI object store.
Note – If a Message Queue client is a J2EE component, JNDI resources are provided by the J2EE container. In such cases, JNDI lookup code may differ from that shown here; see your J2EE provider documentation for details.

EXAMPLE 2–3  Looking Up a Destination

// Create the environment for constructing the initial JNDI naming context.
Hashtable env = new Hashtable();

// Store the environment attributes that tell JNDI which initial context factory to use and where to find the provider.
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, "file:///C:/imq_admin_objects");

// Create the initial context.
Context ctx = new InitialContext(env);

// Look up the destination object in the JNDI object store.
String DEST_LOOKUP_NAME = "MyDest";
Destination MyDest = (Destination) ctx.lookup(DEST_LOOKUP_NAME);

The following section explains the program statements in Example 2–3.

▼ To Look Up a Destination With JNDI

1 Create the environment for constructing the initial JNDI naming context.
How you create the initial context depends on whether you are using a file-system object store or a Lightweight Directory Access Protocol (LDAP) server for your Message Queue administered objects. The code shown here assumes a file-system store; for information about the corresponding LDAP object store attributes, see “LDAP Server Object Stores” in Oracle GlassFish Message Queue 4.4.2 Administration Guide.
The constructor for the initial context accepts an environment parameter, a hash table whose entries specify the attributes for creating the context:

```java
Hashtable env = new Hashtable();
```

You can also set an environment by specifying system properties on the command line, rather than programmatically. For instructions, see the README file in the JMS example applications directory.

2 **Store the environment attributes that tell JNDI which initial context factory to use and where to find the JMS provider.**

The names of these attributes are defined as static constants in class `Context`:

```java
env.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.sun.jndi.fscontext.RefFSContextFactory");
env.put(Context.PROVIDER_URL, "file:///C:/imq_admin_objects");
```

**Note** – The directory represented by `C:/imq_admin_objects` must already exist; if necessary, you must create the directory before referencing it in your code.

3 **Create the initial context.**

```java
Context ctx = new InitialContext(env);
```

If you use system properties to set the environment, omit the environment parameter when creating the context:

```java
Context ctx = new InitialContext();
```

4 **Look up the destination object in the administered object store and typecast it to the appropriate class:**

```java
String DEST_LOOKUP_NAME = "MyDest";
Destination MyDest = (Destination) ctx.lookup(DEST_LOOKUP_NAME);
```

The lookup name you use, `DEST_LOOKUP_NAME`, must match the name used when the object was stored. Note that the actual destination object returned from the object store will always be either a (point-to-point) queue or a (publish/subscribe) topic, but that either can be assigned to a variable of the generic unified-domain class `Destination`.

**Note** – For topic destinations, a symbolic lookup name that includes wildcard characters can be used as the lookup string. See “Supported Topic Destination Names” in Oracle GlassFish Message Queue 4.4.2 Administration Guide.

You can now proceed to send and receive messages using the destination, as described under “Sending Messages” on page 61 and “Receiving Messages” on page 64.
Instantiating a Destination

As with connection factories, you may sometimes find it more convenient to dispense with JNDI lookup and simply create your own queue or topic destination objects by direct instantiation. Although a variable of type Destination can accept objects of either class, you cannot directly instantiate a Destination object; the object must always belong to one of the specific classes Queue or Topic. The constructors for both of these classes accept a string argument specifying the name of the physical destination to which the object corresponds:

```java
Destination myDest = new com.sun.messaging.Queue("myDest");
```

Note, however, that this only creates a Java object representing the destination; it does not actually create a physical destination on the message broker. The physical destination itself must still be created by a Message Queue administrator, with the same name you pass to the constructor when instantiating the object.

**Note** – Destination names beginning with the letters `mq` are reserved and should not be used by client programs.

Also, for topic destinations, a symbolic lookup name that includes wildcard characters can be used as the lookup string. See “Supported Topic Destination Names” in Oracle GlassFish Message Queue 4.4.2 Administration Guide.

Unlike connection factories, destinations have a much more limited set of configuration properties. In fact, only two such properties are defined in the Message Queue class DestinationConfiguration: the name of the physical destination itself (`imqDestinationName`) and an optional descriptive string (`imqDestinationDescription`). Since the latter property is rarely used and the physical destination name can be supplied directly as an argument to the Queue or Topic constructor as shown above, there normally is no need (as there often is with a connection factory) to specify additional properties with the object's `setProperty` method. Hence the variable to which you assign the destination object (`myDest` in the example above) need not be typed with the Message Queue class `com.sun.messaging.Destination`; the standard JMS interface `javax.jms.Destination` (which the Message Queue class implements) is sufficient. If you have imported the full JMS package `javax.jms.*`, you can simply declare the variable with the unqualified name `Destination`, as above, rather than with something like

```java
com.sun.messaging.Destination myDest = new com.sun.messaging.Queue("myDest");
```

as shown earlier for connection factories.
Temporary Destinations

A temporary destination is one that exists only for the duration of the connection that created it. You may sometimes find it convenient to create such a destination to use, for example, as a reply destination for messages you send. Temporary destinations are created with the session method `createTemporaryQueue` or `createTemporaryTopic` (see “Working With Sessions” on page 47 below): for example,

```java
TemporaryQueue tempQueue = mySession.createTemporaryQueue();
```

Although the temporary destination is created by a particular session, its scope is actually the entire connection to which that session belongs. Any of the connection’s sessions (not just the one that created the temporary destination) can create a message consumer for the destination and receive messages from it. The temporary destination is automatically deleted when its connection is closed, or you can delete it explicitly by calling its `delete` method:

```java
tempQueue.delete();
```

Working With Sessions

A session is a single-threaded context for producing and consuming messages. You can create multiple message producers and consumers for a single session, but you are restricted to using them serially, in a single logical thread of control.

Table 2–3 shows the methods defined in the `Session` interface; they are discussed in the relevant sections below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createProducer</td>
<td>Create message producer</td>
</tr>
<tr>
<td>createConsumer</td>
<td>Create message consumer</td>
</tr>
<tr>
<td>createDurableSubscriber</td>
<td>Create durable subscriber for topic</td>
</tr>
<tr>
<td>unsubscribe</td>
<td>Delete durable subscription to topic</td>
</tr>
<tr>
<td>createMessage</td>
<td>Create null message</td>
</tr>
<tr>
<td>createTextMessage</td>
<td>Create text message</td>
</tr>
<tr>
<td>createStreamMessage</td>
<td>Create stream message</td>
</tr>
<tr>
<td>createMapMessage</td>
<td>Create map message</td>
</tr>
<tr>
<td>createObjectMessage</td>
<td>Create object message</td>
</tr>
</tbody>
</table>
### TABLE 2–3  Session Methods  (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createBytesMessage</td>
<td>Create bytes message</td>
</tr>
<tr>
<td>createQueue</td>
<td>Create queue destination</td>
</tr>
<tr>
<td>createTopic</td>
<td>Create topic destination</td>
</tr>
<tr>
<td>createTemporaryQueue</td>
<td>Create temporary queue</td>
</tr>
<tr>
<td>createTemporaryTopic</td>
<td>Create temporary topic</td>
</tr>
<tr>
<td>createBrowser</td>
<td>Create message browser</td>
</tr>
<tr>
<td>setMessageListener</td>
<td>Set distinguished message listener</td>
</tr>
<tr>
<td>getMessageListener</td>
<td>Get distinguished message listener</td>
</tr>
<tr>
<td>getAcknowledgeMode</td>
<td>Get session’s acknowledgment mode</td>
</tr>
<tr>
<td>getTransacted</td>
<td>Is session transacted?</td>
</tr>
<tr>
<td>commit</td>
<td>Commit transaction</td>
</tr>
<tr>
<td>rollback</td>
<td>Roll back transaction</td>
</tr>
<tr>
<td>recover</td>
<td>Recover unacknowledged messages</td>
</tr>
<tr>
<td>close</td>
<td>Close session</td>
</tr>
</tbody>
</table>

Every session exists within the context of a particular connection. The number of sessions you can create for a single connection is limited only by system resources. As described earlier (see “Using Connections” on page 39), you use the connection’s `createSession` method to create a session:

```java
Session mySession = myConnection.createSession(false, Session.AUTO_ACKNOWLEDGE);
```

The first (boolean) argument specifies whether the session is transacted; see “Transacted Sessions” on page 50 for further discussion. The second argument is an integer constant representing the session’s acknowledgment mode, as described in the next section.

## Acknowledgment Modes

A session’s **acknowledgment mode** determines the way your application handles the exchange of acknowledgment information when receiving messages from a broker. The JMS specification defines three possible acknowledgment modes:
In **auto-acknowledge mode**, the Message Queue client runtime immediately sends a *client acknowledgment* for each message it delivers to the message consumer; it then blocks waiting for a return *broker acknowledgment* confirming that the broker has received the client acknowledgment. This acknowledgment “handshake” between client and broker is handled automatically by the client runtime, with no need for explicit action on your part.

In **client-acknowledge mode**, your client application must explicitly acknowledge the receipt of all messages. This allows you to defer acknowledgment until after you have finished processing the message, ensuring that the broker will not delete it from persistent storage before processing is complete. You can either acknowledge each message individually or batch multiple messages and acknowledge them all at once; the client acknowledgment you send to the broker applies to all messages received since the previous acknowledgment. In either case, as in auto-acknowledge mode, the session thread blocks after sending the client acknowledgment, waiting for a broker acknowledgment in return to confirm that your client acknowledgment has been received.

In **dups-OK-acknowledgment mode**, the session automatically sends a client acknowledgment each time it has received a fixed number of messages, or when a fixed time interval has elapsed since the last acknowledgment was sent. (This fixed batch size and timeout interval are currently 10 messages and 7 seconds, respectively, and are not configurable by the client.) Unlike the first two modes described above, the broker does not acknowledge receipt of the client acknowledgment, and the session thread does not block awaiting such return acknowledgment from the broker. This means that you have no way to confirm that your acknowledgment has been received; if it is lost in transmission, the broker may redeliver the same message more than once. However, because client acknowledgments are batched and the session thread does not block, applications that can tolerate multiple delivery of the same message can achieve higher throughput in this mode than in auto-acknowledge or client-acknowledge mode.

Message Queue extends the JMS specification by adding a fourth acknowledgment mode:

In **no-acknowledge mode**, your client application does not acknowledge receipt of messages, nor does the broker expect any such acknowledgment. There is thus no guarantee whatsoever that any message sent by the broker has been successfully received. This mode sacrifices all reliability for the sake of maximum throughput of message traffic.

The standard JMS Session interface, defined in package `javax.jms`, defines static constants for the first three acknowledgment modes (**AUTO_ACKNOWLEDGE**, **CLIENT_ACKNOWLEDGE**, and **DUPS_OK_ACKNOWLEDGE**), to be used as arguments to the connection’s `createSession` method.

The constant representing the fourth mode (**NO_ACKNOWLEDGE**) is defined in the extended Message Queue version of the interface, in package `com.sun.messaging.jms`. The session method `getAcknowledgeMode` returns one of these constants:

```java
int ackMode = mySession.getAcknowledgeMode();
switch (ackMode)
{
    case Session.AUTO_ACKNOWLEDGE:
        /* Code here to handle auto-acknowledge mode */
        break;
```
break;
case Session.CLIENT_ACKNOWLEDGE:
    /* Code here to handle client-acknowledge mode */
    break;
case Session.DUPS_OK_ACKNOWLEDGE:
    /* Code here to handle dups-OK-acknowledge mode */
    break;
case com.sun.messaging.jms.Session.NO_ACKNOWLEDGE:
    /* Code here to handle no-acknowledge mode */
    break;
}

Note – All of the acknowledgment modes discussed above apply to message consumption. For message production, the broker's acknowledgment behavior depends on the message's delivery mode (persistent or nonpersistent; see "Message Header" on page 52). The broker acknowledges the receipt of persistent messages, but not of nonpersistent ones; this behavior is not configurable by the client.

In a transacted session (see next section), the acknowledgment mode is ignored and all acknowledgment processing is handled for you automatically by the Message Queue client runtime. In this case, the getAcknowledgeMode method returns the special constant Session.SESSION_TRANSACTED.

Transacted Sessions

Transactions allow you to group together an entire series of incoming and outgoing messages and treat them as an atomic unit. The message broker tracks the state of the transaction's individual messages, but does not complete their delivery until you commit the transaction. In the event of failure, you can rollback the transaction, canceling all of its messages and restarting the entire series from the beginning.

Transactions always take place within the context of a single session. To use them, you must create a transacted session by passing true as the first argument to a connection's createSession method:

```java
Session
    mySession = myConnection.createSession(true, Session.SESSION_TRANSACTED);
```

The session's getTransacted method tests whether it is a transacted session:

```java
if ( mySession.getTransacted() )
    { /* Code here to handle transacted session */
    }
else
```
A transacted session always has exactly one open transaction, encompassing all messages sent or received since the session was created or the previous transaction was completed. Committing or rolling back a transaction ends that transaction and automatically begins another.

**Note** – Because the scope of a transaction is limited to a single session, it is not possible to combine the production and consumption of a message into a single end-to-end transaction. That is, the delivery of a message from a message producer to a destination on the broker cannot be placed in the same transaction with its subsequent delivery from the destination to a consumer.

When all messages in a transaction have been successfully delivered, you call the session’s `commit` method to commit the transaction:

```java
mySession.commit();
```

All of the session’s incoming messages are acknowledged and all of its outgoing messages are sent. The transaction is then considered complete and a new one is started.

When a send or receive operation fails, an exception is thrown. While it is possible to handle the exception by simply ignoring it or by retrying the operation, it is recommended that you roll back the transaction, using the session’s `rollback` method:

```java
mySession.rollback();
```

All of the session’s incoming messages are recovered and redelivered, and its outgoing messages are destroyed and must be re-sent.

**Working With Messages**

This section describes how to use the Message Queue Java API to compose, send, receive, and process messages.

**Message Structure**

A message consists of the following parts:

- A header containing identifying and routing information
- Optional properties that can be used to convey additional identifying information beyond that contained in the header
A body containing the actual content of the message

The following sections discuss each of these in greater detail.

**Message Header**

Every message must have a header containing identifying and routing information. The header consists of a set of standard fields, which are defined in the *Java Message Service Specification* and summarized in Table 2–4. Some of these are set automatically by Message Queue in the course of producing and delivering a message, some depend on settings specified when a message producer sends a message, and others are set by the client on a message-by-message basis.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSMessageID</td>
<td>Message identifier</td>
</tr>
<tr>
<td>JMSDestination</td>
<td>Destination to which message is sent</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td>Destination to which to reply</td>
</tr>
<tr>
<td>JMSCorrelationID</td>
<td>Link to related message</td>
</tr>
<tr>
<td>JMSDeliveryMode</td>
<td>Delivery mode (persistent or nonpersistent)</td>
</tr>
<tr>
<td>JMSPriority</td>
<td>Priority level</td>
</tr>
<tr>
<td>JMSTimestamp</td>
<td>Time of transmission</td>
</tr>
<tr>
<td>JMSExpiration</td>
<td>Expiration time</td>
</tr>
<tr>
<td>JMSType</td>
<td>Message type</td>
</tr>
<tr>
<td>JMSRedelivered</td>
<td>Has message been delivered before?</td>
</tr>
</tbody>
</table>

The JMS Message interface defines methods for setting the value of each header field: for instance,

```java
tomcat.outMsg.setJMSReplyTo(replyDest);
```

Table 2–5 lists all of the available header specification methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setJMSMessageID</td>
<td>Set message identifier</td>
</tr>
<tr>
<td>setJMSDestination</td>
<td>Set destination</td>
</tr>
</tbody>
</table>
### Table 2-5  Message Header Specification Methods  (Continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setJMSReplyTo</td>
<td>Set reply destination</td>
</tr>
<tr>
<td>setJMSCorrelationID</td>
<td>Set correlation identifier from string</td>
</tr>
<tr>
<td>setJMSCorrelationIDAsBytes</td>
<td>Set correlation identifier from byte array</td>
</tr>
<tr>
<td>setJMSDeliveryMode</td>
<td>Set delivery mode</td>
</tr>
<tr>
<td>setJMSPriority</td>
<td>Set priority level</td>
</tr>
<tr>
<td>setJMSTimestamp</td>
<td>Set time stamp</td>
</tr>
<tr>
<td>setJMSExpiration</td>
<td>Set expiration time</td>
</tr>
<tr>
<td>setJMSType</td>
<td>Set message type</td>
</tr>
<tr>
<td>setJMSRedelivered</td>
<td>Set redelivered flag</td>
</tr>
</tbody>
</table>

The **message identifier** (`JMSMessageID`) is a string value uniquely identifying the message, assigned and set by the message broker when the message is sent. Because generating an identifier for each message adds to both the size of the message and the overhead involved in sending it, and because some client applications may not use them, the JMS interface provides a way to suppress the generation of message identifiers, using the message producer method `setDisableMessageID` (see “Sending Messages” on page 61).

The **JMSDestination** header field holds a `Destination` object representing the destination to which the message is directed, set by the message broker when the message is sent. There is also a `JMSReplyTo` field that you can set to specify a destination to which reply messages should be directed. Clients sending such a reply message can set its `JMSCorrelationID` header field to refer to the message to which they are replying. Typically this field is set to the message identifier string of the message being replied to, but client applications are free to substitute their own correlation conventions instead, using either the `setJMSCorrelationID` method (if the field value is a string) or the more general `setJMSCorrelationIDAsBytes` (if it is not).

The **delivery mode** (`JMSDeliveryMode`) specifies whether the message broker should log the message to stable storage. There are two possible values, `PERSISTENT` and `NON_PERSISTENT`, both defined as static constants of the JMS interface `DeliveryMode`: for example,

```java
outMsg.setJMSDeliveryMode(DeliveryMode.NON_PERSISTENT);
```

The default delivery mode is `PERSISTENT`, represented by the static constant `Message.DEFAULT_DELIVERY_MODE`.

The choice of delivery mode represents a tradeoff between performance and reliability:

- **In persistent mode**, the broker logs the message to stable storage, ensuring that it will not be lost in transit in the event of transmission failure; the message is guaranteed to be delivered exactly once.
In nonpersistent mode, the message is not logged to stable storage; it will be delivered at most once, but may be lost in case of failure and not delivered at all. This mode does, however, improve performance by reducing the broker's message-handling overhead. It may thus be appropriate for applications in which performance is at a premium and reliability is not.

The message's priority level (JMSPriority) is expressed as an integer from 0 (lowest) to 9 (highest). Priorities from 0 to 4 are considered gradations of normal priority, those from 5 to 9 of expedited priority. The default priority level is 4, represented by the static constant Message.DEFAULT_PRIORITY.

The Message Queue client runtime sets the JMSStamp header field to the time it delivers the message to the broker, expressed as a long integer in standard Java format (milliseconds since midnight, January 1, 1970 UTC). The message's lifetime, specified when the message is sent, is added to this value and the result is stored in the JMSExpiration header field. (The default lifetime value of 0, represented by the static constant Message.DEFAULT_TIME_TO_LIVE, denotes an unlimited lifetime. In this case, the expiration time is also set to 0 to indicate that the message never expires.) As with the message identifier, client applications that do not use a message's time stamp can improve performance by suppressing its generation with the message producer method setDisableMessageTimestamp (see “Sending Messages” on page 61).

The header field JMSType can contain an optional message type identifier string supplied by the client when the message is sent. This field is intended for use with other JMS providers; Message Queue clients can simply ignore it.

When a message already delivered must be delivered again because of a failure, the broker indicates this by setting the JMSRedelivered flag in the message header to true. This can happen, for instance, when a session is recovered or a transaction is rolled back. The receiving client can check this flag to avoid duplicate processing of the same message (such as when the message has already been successfully received but the client's acknowledgment was missed by the broker).

See the Java Message Service Specification for a more detailed discussion of all message header fields.

Message Properties

A message property consists of a name string and an associated value, which must be either a string or one of the standard Java primitive data types (int, byte, short, long, float, double, or boolean). The Message interface provides methods for setting properties of each type (see Table 2–6). There is also a setObjectProperty method that accepts a primitive value in objectified form, as a Java object of class Integer, Byte, Short, Long, Float, Double, Boolean, or String. The clearProperties method deletes all properties associated with a message; the message header and body are not affected.
The JMS specification defines certain standard properties, listed in Table 2–7. By convention, the names of all such standard properties begin with the letters JMSX; names of this form are reserved and must not be used by a client application for its own custom message properties. Similarly, property names beginning with JMS_SUN are reserved for provider-specific properties defined by Message Queue itself; these are discussed in Chapter 3, “Message Queue Clients: Design and Features”.

### TABLE 2–6 Message Property Specification Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setIntProperty</td>
<td>Set integer property</td>
</tr>
<tr>
<td>setByteProperty</td>
<td>Set byte property</td>
</tr>
<tr>
<td>setShortProperty</td>
<td>Set short integer property</td>
</tr>
<tr>
<td>setLongProperty</td>
<td>Set long integer property</td>
</tr>
<tr>
<td>setFloatProperty</td>
<td>Set floating-point property</td>
</tr>
<tr>
<td>setDoubleProperty</td>
<td>Set double-precision property</td>
</tr>
<tr>
<td>setBooleanProperty</td>
<td>Set boolean property</td>
</tr>
<tr>
<td>setStringProperty</td>
<td>Set string property</td>
</tr>
<tr>
<td>setObjectProperty</td>
<td>Set property from object</td>
</tr>
<tr>
<td>clearProperties</td>
<td>Clear properties</td>
</tr>
</tbody>
</table>

### TABLE 2–7 Standard JMS Message Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSXUserID</td>
<td>Identity of user sending message</td>
</tr>
<tr>
<td>JMSXAppID</td>
<td>Identity of application sending message</td>
</tr>
<tr>
<td>JMSXDeliveryCount</td>
<td>Number of delivery attempts</td>
</tr>
<tr>
<td>JMSXGroupID</td>
<td>Identity of message group to which this message belongs</td>
</tr>
<tr>
<td>JMSXGroupSeq</td>
<td>Sequence number within message group</td>
</tr>
<tr>
<td>JMSXProducerTXID</td>
<td>Identifier of transaction within which message was produced</td>
</tr>
<tr>
<td>JMSXConsumerTXID</td>
<td>Identifier of transaction within which message was consumed</td>
</tr>
<tr>
<td>JMSXRcvTimestamp</td>
<td>Time message delivered to consumer</td>
</tr>
<tr>
<td>JMSXState</td>
<td>Message state (waiting, ready, expired, or retained)</td>
</tr>
</tbody>
</table>
Message Body

The actual content of a message is contained in the *message body*. JMS defines six classes (or types) of message, each with a different body format:

- A **text message** (interface `TextMessage`) contains a Java string.
- A **stream message** (interface `StreamMessage`) contains a stream of Java primitive values, written and read sequentially.
- A **map message** (interface `MapMessage`) contains a set of name-value pairs, where each name is a string and each value is a Java primitive value. The order of the entries is undefined; they can be accessed randomly by name or enumerated sequentially.
- An **object message** (interface `ObjectMessage`) contains a serialized Java object (which may in turn be a collection of other objects).
- A **bytes message** (interface `BytesMessage`) contains a stream of uninterpreted bytes.
- A **null message** (interface `Message`) consists of a header and properties only, with no message body.

Each of these is a subinterface of the generic `Message` interface, extended with additional methods specific to the particular message type.

Composing Messages

The JMS `Session` interface provides methods for creating each type of message, as shown in Table 2–8. For instance, you can create a text message with a statement such as

```
TextMessage outMsg = mySession.createTextMessage();
```

In general, these methods create a message with an empty body; the interfaces for specific message types then provide additional methods for filling the body with content, as described in the sections that follow.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createMessage</code></td>
<td>Create null message</td>
</tr>
<tr>
<td><code>createTextMessage</code></td>
<td>Create text message</td>
</tr>
<tr>
<td><code>createStreamMessage</code></td>
<td>Create stream message</td>
</tr>
<tr>
<td><code>createMapMessage</code></td>
<td>Create map message</td>
</tr>
<tr>
<td><code>createObjectMessage</code></td>
<td>Create object message</td>
</tr>
<tr>
<td><code>createBytesMessage</code></td>
<td>Create bytes message</td>
</tr>
</tbody>
</table>
Note – Some of the message-creation methods have an overloaded form that allows you to initialize the message body directly at creation: for example,

```java
TextMessage
    outMsg = mySession.createTextMessage("Hello, World!");
```

These exceptions are pointed out in the relevant sections below.

Once a message has been delivered to a message consumer, its body is considered read-only; any attempt by the consumer to modify the message body will cause an exception (MessageNotWriteableException) to be thrown. The consumer can, however, empty the message body and place it in a writeable state by calling the message method `clearBody`:

```java
outMsg.clearBody();
```

This places the message in the same state as if it had been newly created, ready to fill its body with new content.

**Composing Text Messages**

You create a text message with the session method `createTextMessage`. You can either initialize the message body directly at creation time

```java
TextMessage outMsg = mySession.createTextMessage("Hello, World!");
```

or simply create an empty message and then use its `setText` method (see Table 2–9 ) to set its content:

```java
TextMessage outMsg = mySession.createTextMessage();
outMsg.setText("Hello, World!");
```

**TABLE 2–9  Text Message Composition Method**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>setText</code></td>
<td>Set content string</td>
</tr>
</tbody>
</table>

**Composing Stream Messages**

The session method `createStreamMessage` returns a new, empty stream message. You can then use the methods shown in Table 2–10 to write primitive data values into the message body, similarly to writing to a data stream: for example,

```java
StreamMessage outMsg = mySession.createStreamMessage();
outMsg.writeString("The Meaning of Life");
outMsg.writeInt(42);
```
As a convenience for handling values whose types are not known until execution time, the `writeObject` method accepts a string or an objectified primitive value of class `Integer`, `Byte`, `Short`, `Long`, `Float`, `Double`, `Boolean`, or `Character` and writes the corresponding string or primitive value to the message stream: for example, the statements

```java
Integer meaningOfLife = new Integer(42);
outMsg.writeObject(meaningOfLife);
```

are equivalent to

```java
outMsg.writeInt(42);
```

This method will throw an exception (`MessageFormatException`) if the argument given to it is not of class `String` or one of the objectified primitive classes.

Once you’ve written the entire message contents to the stream, the `reset` method

```java
outMsg.reset();
```

puts the message body in read-only mode and repositions the stream to the beginning, ready to read (see “Processing Messages” on page 71). When the message is in this state, any attempt to write to the message stream will throw the exception `MessageNotWriteableException`. A call to the `clearBody` method (inherited from the superinterface `Message`) deletes the entire message body and makes it writeable again.
Composing Map Messages

Table 2–11 shows the methods available in the MapMessage interface for adding content to the body of a map message. Each of these methods takes two arguments, a name string and a primitive or string value of the appropriate type, and adds the corresponding name-value pair to the message body: for example,

```java
StreamMessage outMsg = mySession.createMapMessage();
outMsg.setInt("The Meaning of Life", 42);
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setInt</td>
<td>Store integer in message map by name</td>
</tr>
<tr>
<td>setByte</td>
<td>Store byte value in message map by name</td>
</tr>
<tr>
<td>setBytes</td>
<td>Store byte array in message map by name</td>
</tr>
<tr>
<td>setShort</td>
<td>Store short integer in message map by name</td>
</tr>
<tr>
<td>setLong</td>
<td>Store long integer in message map by name</td>
</tr>
<tr>
<td>setFloat</td>
<td>Store floating-point value in message map by name</td>
</tr>
<tr>
<td>setDouble</td>
<td>Store double-precision value in message map by name</td>
</tr>
<tr>
<td>setBoolean</td>
<td>Store boolean value in message map by name</td>
</tr>
<tr>
<td>setChar</td>
<td>Store character in message map by name</td>
</tr>
<tr>
<td>setString</td>
<td>Store string in message map by name</td>
</tr>
<tr>
<td>setObject</td>
<td>Store object in message map by name</td>
</tr>
</tbody>
</table>

Like stream messages, map messages provide a convenience method (setObject) for dealing with values whose type is determined dynamically at execution time: for example, the statements

```java
Integer meaningOfLife = new Integer(42);
outMsg.setObject("The Meaning of Life", meaningOfLife);
```

are equivalent to

```java
outMsg.setInt("The Meaning of Life", 42);
```

The object supplied must be either a string object (class String) or an objectified primitive value of class Integer, Byte, Short, Long, Float, Double, Boolean, or Character; otherwise an exception (MessageFormatException) will be thrown.
Composing Object Messages

The ObjectMessage interface provides just one method, setObject (Table 2–12), for setting the body of an object message:

```java
ObjectMessage outMsg = mySession.createObjectMessage();
outMsg.setObject(bodyObject);
```

The argument to this method can be any serializable object (that is, an instance of any class that implements the standard Java interface `Serializable`). If the object is not serializable, the exception `MessageFormatException` will be thrown.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setObject</td>
<td>Serialize object to message body</td>
</tr>
</tbody>
</table>

As an alternative, you can initialize the message body directly when you create the message, by passing an object to the session method `createObjectMessage`:

```java
ObjectMessage outMsg = mySession.createObjectMessage(bodyObject);
```

Again, an exception will be thrown if the object is not serializable.

Composing Bytes Messages

The body of a bytes message simply consists of a stream of uninterpreted bytes; its interpretation is entirely a matter of agreement between sender and receiver. This type of message is intended primarily for encoding message formats required by other existing message systems; Message Queue clients should generally use one of the other, more specific message types instead.

Composing a bytes message is similar to composing a stream message (see “Composing Stream Messages” on page 57). You create the message with the session method `createBytesMessage`, then use the methods shown in Table 2–13 to encode primitive values into the message’s byte stream: for example,

```java
BytesMessage outMsg = mySession.createBytesMessage();
outMsg.writeUTF("The Meaning of Life");
outMsg.writeInt(42);
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeInt</td>
<td>Write integer to message stream</td>
</tr>
</tbody>
</table>

Working With Messages
As with stream and map messages, you can use the generic object-based method `writeObject` to handle values whose type is unknown at compilation time: for example, the statements

```
Integer meaningOfLife = new Integer(42);
outMsg.writeObject(meaningOfLife);
```

are equivalent to

```
outMsg.writeInt(42);
```

The message's `reset` method

```
outMsg.reset();
```

puts the message body in read-only mode and repositions the byte stream to the beginning, ready to read (see "Processing Messages" on page 71). Attempting to write further content to a message in this state will cause an exception (`MessageNotWriteableException`). The inherited `Message` method `clearBody` can be used to delete the entire message body and make it writeable again.

## Sending Messages

In order to send messages to a message broker, you must create a *message producer* using the session method `createProducer`:
MessageProducer myProducer = mySession.createProducer(myDest);

The scope of the message producer is limited to the session that created it and the connection to which that session belongs. Table 2–14 shows the methods defined in the MessageProducer interface.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getDestination</td>
<td>Get default destination</td>
</tr>
<tr>
<td>setDeliveryMode</td>
<td>Set default delivery mode</td>
</tr>
<tr>
<td>getDeliveryMode</td>
<td>Get default delivery mode</td>
</tr>
<tr>
<td>setPriority</td>
<td>Set default priority level</td>
</tr>
<tr>
<td>getPriority</td>
<td>Get default priority level</td>
</tr>
<tr>
<td>setTimeToLive</td>
<td>Set default message lifetime</td>
</tr>
<tr>
<td>getTimeToLive</td>
<td>Get default message lifetime</td>
</tr>
<tr>
<td>setDisableMessageID</td>
<td>Set message identifier disable flag</td>
</tr>
<tr>
<td>getDisableMessageID</td>
<td>Get message identifier disable flag</td>
</tr>
<tr>
<td>setDisableMessageTimestamp</td>
<td>Set time stamp disable flag</td>
</tr>
<tr>
<td>getDisableMessageTimestamp</td>
<td>Get time stamp disable flag</td>
</tr>
<tr>
<td>send</td>
<td>Send message</td>
</tr>
<tr>
<td>close</td>
<td>Close message producer</td>
</tr>
</tbody>
</table>

The createProducer method takes a destination as an argument, which may be either a (point-to-point) queue or a (publish/subscribe) topic. The producer will then send all of its messages to the specified destination. If the destination is a queue, the producer is called a sender for that queue; if it is a topic, the producer is a publisher to that topic. The message producer’s getDestination method returns this destination.

You also have the option of leaving the destination unspecified when you create a producer

MessageProducer myProducer = mySession.createProducer(null);

in which case you must specify an explicit destination for each message. This option is typically used for producers that must send messages to a variety of destinations, such as those designated in the JMSReplyTo header fields of incoming messages (see “Message Header” on page 52).
The generic `MessageProducer` interface also has specialized subinterfaces, `QueueSender` and `TopicPublisher`, for sending messages specifically to a point-to-point queue or a publish/subscribe topic. These types of producer are created by the `createSender` and `createPublisher` methods of the specialized session subinterfaces `QueueSession` and `TopicSession`, respectively. However, it is generally more convenient (and recommended) to use the generic form of message producer described here, which can handle both types of destination indiscriminately.

A producer has a default delivery mode (persistent or nonpersistent), priority level, and message lifetime, which it will apply to all messages it sends unless explicitly overridden for an individual message. You can set these properties with the message producer methods `setDeliveryMode`, `setPriority`, and `setTimeToLive`, and retrieve them with `getDeliveryMode`, `getPriority`, and `getTimeToLive`. If you don’t set them explicitly, they default to persistent delivery, priority level 4, and a lifetime value of 0, denoting an unlimited message lifetime.

The heart of the message producer interface is the `send` method, which is available in a variety of overloaded forms. The simplest of these just takes a message as its only argument:

```java
myProducer.send(outMsg);
```

This sends the specified message to the producer’s default destination, using the producer’s default delivery mode, priority, and message lifetime. Alternatively, you can explicitly specify the destination

```java
myProducer.send(myDest, outMsg);
```

or the delivery mode, priority, and lifetime in milliseconds

```java
myProducer.send(outMsg, DeliveryMode.NON_PERSISTENT, 9, 1000);
```

or all of these at once:

```java
myProducer.send(myDest, outMsg, DeliveryMode.NON_PERSISTENT, 9, 1000);
```

Recall that if you did not specify a destination when creating the message producer, you must provide an explicit destination for each message you send.

As discussed earlier under “Message Header” on page 52, client applications that have no need for the message identifier and time stamp fields in the message header can gain some performance improvement by suppressing the generation of these fields, using the message producer’s `setDisableMessageID` and `setdisableMessageTimestamp` methods. Note that a true value for either of these flags disables the generation of the corresponding header field, while a false value enables it. Both flags are set to false by default, meaning that the broker will generate the values of these header fields unless explicitly instructed otherwise.
When you are finished using a message producer, you should call its `close` method

```java
myProducer.close();
```

allowing the broker and client runtime to release any resources they may have allocated on the producer's behalf.

**Receiving Messages**

Messages are received by a *message consumer*, within the context of a connection and a session. Once you have created a consumer, you can use it to receive messages in either of two ways:

- **In synchronous message consumption**, you explicitly request the delivery of messages when you are ready to receive them.
- **In asynchronous message consumption**, you register a *message listener* for the consumer. The Message Queue client runtime then calls the listener whenever it has a message to deliver.

These two forms of message consumption are described in the sections "Receiving Messages Synchronously" on page 67 and "Receiving Messages Asynchronously" on page 67.

**Creating Message Consumers**

The session method `createConsumer` creates a generic consumer that can be used to receive messages from either a (point-to-point) queue or a (publish/subscribe) topic:

```java
MessageConsumer myConsumer = mySession.createConsumer(myDest);
```

If the destination is a queue, the consumer is called a *receiver* for that queue; if it is a topic, the consumer is a *subscriber to that topic*.

**Note** – The generic `MessageConsumer` interface also has specialized subinterfaces, `QueueReceiver` and `TopicSubscriber`, for receiving messages specifically from a point-to-point queue or a publish/subscribe topic. These types of consumer are created by the `createReceiver` and `createSubscriber` methods of the specialized session subinterfaces `QueueSession` and `TopicSession`, respectively. However, it is generally more convenient (and recommended) to use the generic form of message consumer described here, which can handle both types of destination indiscriminately.

A subscriber created for a topic destination with the `createConsumer` method is always *nondurable*, meaning that it will receive only messages that are sent (*published*) to the topic while the subscriber is active. If you want the broker to retain messages published to a topic while no subscriber is active and deliver them when one becomes active again, you must instead create a *durable subscriber*, as described in "Durable Subscribers" on page 66.
Table 2–15 shows the methods defined in the MessageConsumer interface, which are discussed in detail in the relevant sections below.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getMessageSelector</td>
<td>Get message selector</td>
</tr>
<tr>
<td>receive</td>
<td>Receive message synchronously</td>
</tr>
<tr>
<td>receiveNoWait</td>
<td>Receive message synchronously without blocking</td>
</tr>
<tr>
<td>setMessageListener</td>
<td>Set message listener for asynchronous reception</td>
</tr>
<tr>
<td>getMessageListener</td>
<td>Get message listener for asynchronous reception</td>
</tr>
<tr>
<td>close</td>
<td>Close message consumer</td>
</tr>
</tbody>
</table>

**Message Selectors**

If appropriate, you can restrict the messages a consumer will receive from its destination by supplying a *message selector* as an argument when you create the consumer:

```java
String mySelector = "/* Text of selector here */";
MessageConsumer myConsumer = mySession.createConsumer(myDest, mySelector);
```

The selector is a string whose syntax is based on a subset of the SQL92 conditional expression syntax, which allows you to filter the messages you receive based on the values of their properties (see "Message Properties" on page 54). See the Java Message Service Specification for a complete description of this syntax. The message consumer's `getMessageSelector` method returns the consumer's selector string (or `null` if no selector was specified when the consumer was created):

```java
String mySelector = myConsumer.getMessageSelector();
```

**Note** – Messages whose properties do not satisfy the consumer's selector will be retained undelivered by the destination until they are retrieved by another message consumer. The use of message selectors can thus cause messages to be delivered out of sequence from the order in which they were originally produced. Only a message consumer without a selector is guaranteed to receive messages in their original order.

In some cases, the same connection may both publish and subscribe to the same topic destination. The `createConsumer` method accepts an optional boolean argument that suppresses the delivery of messages published by the consumer's own connection:
String mySelector = "/* Text of selector here */";
MessageConsumer
   myConsumer = mySession.createConsumer(myDest, mySelector, true);

The resulting consumer will receive only messages published by a different connection.

Durable Subscribers

To receive messages delivered to a publish/subscribe topic while no message consumer is active, you must ask the message broker to create a durable subscriber for that topic. All sessions that create such subscribers for a given topic must have the same client identifier (see "Using Connections" on page 39). When you create a durable subscriber, you supply a subscriber name that must be unique for that client identifier:

MessageConsumer
   myConsumer = mySession.createDurableSubscriber(myDest, "mySub");

(The object returned by the createDurableSubscriber method is actually typed as TopicSubscriber, but since that is a subinterface of MessageConsumer, you can safely assign it to a MessageConsumer variable. Note, however, that the destination myDest must be a publish/subscribe topic and not a point-to-point queue.)

You can think of a durable subscriber as a "virtual message consumer" for the specified topic, identified by the unique combination of a client identifier and subscriber name. When a message arrives for the topic and no message consumer is currently active for it, the message will be retained for later delivery. Whenever you create a consumer with the given client identifier and subscriber name, it will be considered to represent this same durable subscriber and will receive all of the accumulated messages that have arrived for the topic in the subscriber's absence. Each message is retained until it is delivered to (and acknowledged by) such a consumer or until it expires.

Note – Only one session at a time can have an active consumer for a given durable subscription. If another such consumer already exists, the createDurableSubscriber method will throw an exception.

Like the createConsumer method described in the preceding section (which creates nondurable subscribers), createDurableSubscriber can accept an optional message selector string and a boolean argument telling whether to suppress the delivery of messages published by the subscriber's own connection:

String mySelector = "/* Text of selector here */";
MessageConsumer myConsumer
   = mySession.createDurableSubscriber(myDest, "mySub",
   mySelector, true);
You can change the terms of a durable subscription by creating a new subscriber with the same
client identifier and subscription name but with a different topic, selector, or both. The effect is
as if the old subscription were destroyed and a new one created with the same name. When you
no longer need a durable subscription, you can destroy it with the session method unsubscribe:

```java
mySession.unsubscribe("mySub");
```

**Receiving Messages Synchronously**

Once you have created a message consumer for a session, using either the createConsumer
or createDurableSubscriber method, you must `start` the session's connection to begin the flow
of incoming messages:

```java
myConnection.start();
```

(Notethat it is not necessary to start a connection in order to produce messages, only to
consume them.) You can then use the consumer's `receive` method to receive messages
synchronously from the message broker:

```java
Message inMsg = myConsumer.receive();
```

This returns the next available message for this consumer. If no message is immediately
available, the `receive` method blocks until one arrives. You can also provide a timeout interval
in milliseconds:

```java
Message inMsg = myConsumer.receive(1000);
```

In this case, if no message arrives before the specified timeout interval (1 second in the example)
expires, the method will return with a null result. An alternative method, `receiveNoWait`,
returns a null result immediately if no message is currently available:

```java
Message inMsg = myConsumer.receiveNoWait();
```

**Receiving Messages Asynchronously**

If you want your message consumer to receive incoming messages asynchronously, you must
create a `message listener` to process the messages. This is a Java object that implements the JMS
MessageListener interface. The procedure is as follows:
To Set Up a Message Queue Java Client to Receive Messages Asynchronously

1 Define a message listener class implementing the MessageListener interface.
   The interface consists of the single method onMessage, which accepts a message as a parameter
   and processes it in whatever way is appropriate for your application:
   ```java
   public class MyMessageListener implements MessageListener {
       public void onMessage (Message inMsg) {
           /* Code here to process message */
       }
   }
   ```

2 Create a message consumer.
   You can use either the createConsumer or createDurableSubscriber method of the session in
   which the consumer will operate: for instance,
   ```java
   MessageConsumer myConsumer = mySession.createConsumer(myDest);
   ```

3 Create an instance of your message listener class.
   ```java
   MyMessageListener myListener = new MyMessageListener();
   ```

4 Associate the message listener with your message consumer.
   The message consumer method setMessageListener accepts a message listener object and
   associates it with the given consumer:
   ```java
   myConsumer.setMessageListener(myListener);
   ```

5 Start the connection to which this consumer's session belongs.
   The connection's start method begins the flow of messages from the message broker to your
   message consumer:
   ```java
   myConnection.start();
   ```
   Once the connection is started, the Message Queue client runtime will call your message
   listener's onMessage method each time it has a message to deliver to this consumer.
   To ensure that no messages are lost before your consumer is ready to receive them, it is
   important not to start the connection until after you have created the message listener and
   associated it with the consumer. If the connection is already started, you should stop it before
   creating an asynchronous consumer, then start it again when the consumer is ready to begin
   processing.
Setting a consumer's message listener to null removes any message listener previously associated with it:

```java
myConsumer.setMessageListener(null);
```

The consumer's `getMessageListener` method returns its current message listener (or null if there is none):

```java
MyMessageListener myListener = myConsumer.getMessageListener();
```

## Acknowledging Messages

If you have specified client-acknowledge as your session's acknowledgment mode (see "Acknowledgment Modes" on page 48), it is your client application's responsibility to explicitly acknowledge each message it receives. If you have received the message synchronously, using a message consumer's `receive` (or `receiveNowait`) method, you should process the message first and then acknowledge it; if you have received it asynchronously, your message listener's `onMessage` method should acknowledge the message after processing it. This ensures that the message broker will not delete the message from persistent storage until processing is complete.

**Note** – In a transacted session (see "Transacted Sessions" on page 50), there is no need to acknowledge a message explicitly: the session's acknowledgment mode is ignored and all acknowledgment processing is handled for you automatically by the Message Queue client runtime. In this case, the session's `getAcknowledgeMode` method will return the special constant `Session.SESSION_TRANSACTED`.

Table 2-16 shows the methods available for acknowledging a message. The most general is `acknowledge`, defined in the standard JMS interface `javax.jms.Message`:

```java
inMsg.acknowledge();
```

This acknowledges all unacknowledged messages consumed by the session up to the time of call. You can use this method to acknowledge each message individually as you receive it, or you can group several messages together and acknowledge them all at once by calling `acknowledge` on the last one in the group.

**Table 2-16 Message Acknowledgment Methods**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acknowledge</code></td>
<td>Acknowledge all unacknowledged messages for session</td>
</tr>
<tr>
<td><code>acknowledgeThisMessage</code></td>
<td>Acknowledge this message only</td>
</tr>
<tr>
<td><code>acknowledgeUpThroughThisMessage</code></td>
<td>Acknowledge all unacknowledged messages through this one</td>
</tr>
</tbody>
</table>
The Message Queue version of the Message interface, defined in the package `com.sun.messaging.jms`, adds two more methods that provide more flexible control over which messages you acknowledge. The `acknowledgeThisMessage` method just acknowledges the single message for which it is called, rather than all messages consumed by the session; `acknowledgeUpThroughThisMessage` acknowledges the message for which it is called and all previous messages; messages received after that message remain unacknowledged.

**Browsing Messages**

If the destination from which you are consuming messages is a point-to-point queue, you can use a *queue browser* to examine the messages in the queue without consuming them. The session method `createBrowser` creates a browser for a specified queue:

```java
QueueBrowser myBrowser = mySession.createBrowser(myDest);
```

The method will throw an exception (`InvalidDestinationException`) if you try to pass it a topic destination instead of a queue. You can also supply a selector string as an optional second argument:

```java
String mySelector = "/* Text of selector here */";
QueueBrowser myBrowser = mySession.createBrowser(myDest, mySelector);
```

Table 2–17 shows the methods defined in the `QueueBrowser` interface. The `getQueue` and `getMessageSelector` methods return the browser's queue and selector string, respectively.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getQueue</td>
<td>Get queue from which this browser reads</td>
</tr>
<tr>
<td>getMessageSelector</td>
<td>Get message selector</td>
</tr>
<tr>
<td>getEnumeration</td>
<td>Get enumeration of all messages in the queue</td>
</tr>
<tr>
<td>close</td>
<td>Close browser</td>
</tr>
</tbody>
</table>

The most important queue browser method is `getEnumeration`, which returns a Java enumeration object that you can use to iterate through the messages in the queue, as shown in Example 2–4.

**EXAMPLE 2–4  Browsing a Queue**

```java
Enumeration queueMessages = myBrowser.getEnumeration();
Message eachMessage;

while ( queueMessages.hasMoreElements() )
{ eachMessage = queueMessages.nextElement();
```
/* Do something with the message */
}

The browser's close method closes it when you're through with it:

myBrowser.close();

**Closing a Consumer**

As a matter of good programming practice, you should close a message consumer when you have no further need for it. Closing a session or connection automatically closes all consumers associated with it; to close a consumer without closing the session or connection to which it belongs, you can use its close method:

myConsumer.close();

For a consumer that is a nondurable topic subscriber, this terminates the flow of messages to the consumer. However, if the consumer is a queue receiver or a durable topic subscriber, messages will continue to be accumulated for the destination and will be delivered the next time a consumer for that destination becomes active. To terminate a durable subscription permanently, call its session's unsubscribe method with the subscriber name as an argument:

mySession.unsubscribe("mySub");

**Processing Messages**

Processing a message after you have received it may entail examining its header fields, properties, and body. The following sections describe how this is done.

**Retrieving Message Header Fields**

The standard JMS message header fields are described in Table 2–4. Table 2–18 shows the methods provided by the JMS Message interface for retrieving the values of these fields: for instance, you can obtain a message's reply destination with the statement

```java
Destination replyDest = inMsg.getJMSReplyTo();
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getJMSMessageID</td>
<td>Get message identifier</td>
</tr>
</tbody>
</table>
Table 2–18 lists the methods defined in the JMS Message interface for retrieving the values of a message's properties (see “Message Properties” on page 54). There is a retrieval method for each of the possible primitive types that a property value can assume: for instance, you can obtain a message's time stamp with the statement

```java
long timeStamp = inMsg.getLongProperty("JMSRcvTimestamp");
```

TABLE 2–19 Message Property Retrieval Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getIntProperty</td>
<td>Get integer property</td>
</tr>
<tr>
<td>getByteProperty</td>
<td>Get byte property</td>
</tr>
<tr>
<td>getShortProperty</td>
<td>Get short integer property</td>
</tr>
<tr>
<td>getLongProperty</td>
<td>Get long integer property</td>
</tr>
<tr>
<td>getFloatProperty</td>
<td>Get floating-point property</td>
</tr>
<tr>
<td>getDoubleProperty</td>
<td>Get double-precision property</td>
</tr>
<tr>
<td>getBooleanProperty</td>
<td>Get boolean property</td>
</tr>
<tr>
<td>getStringProperty</td>
<td>Get string property</td>
</tr>
<tr>
<td>getObjectProperty</td>
<td>Get property as object</td>
</tr>
</tbody>
</table>
There is also a generic `getObjectProperty` method that returns a property value in objectified form, as a Java object of class `Integer`, `Byte`, `Short`, `Long`, `Float`, `Double`, `Boolean`, or `String`. For example, another way to obtain a message's time stamp, equivalent to that shown above, would be

```java
Long timeStampObject = (Long)inMsg.getObjectProperty("JMSXRsTimestamp");
long timeStamp = timeStampObject.longValue();
```

If the message has no property with the requested name, `getObjectProperty` will return `null`; the message method `propertyExists` tests whether this is the case.

The `getPropertyNames` method returns a Java enumeration object for iterating through all of the property names associated with a given message; you can then use the retrieval methods shown in the table to retrieve each of the properties by name, as shown in Example 2-5.

### Example 2-5  Enumerating Message Properties

```
Enumeration propNames = inMsg.getPropertyNames();
String eachName;
Object eachValue;

while ( propNames.hasMoreElements() )
{ eachName = propNames.nextElement();
  eachValue = inMsg.getObjectProperty(eachName);
  /* Do something with the value */
}
```

### Processing the Message Body

The methods for retrieving the contents of a message's body essentially parallel those for composing the body, as described earlier under “Composing Messages” on page 56. The following sections describe these methods for each of the possible message types (text, stream, map, object, and bytes).

#### Processing Text Messages

The text message method `getText` (Table 2-20 ) retrieves the contents of a text message's body in the form of a string:

```java
String textBody = inMsg.getText();
```
TABLE 2–20  Text Message Access Method

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getText</td>
<td>Get content string</td>
</tr>
</tbody>
</table>

**Processing Stream Messages**

Reading the contents of a stream message is similar to reading from a data stream, using the access methods shown in Table 2–21: for example, the statement

```java
int intVal = inMsg.readInt();
```

retrieves an integer value from the message stream.

TABLE 2–21  Stream Message Access Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readInt</td>
<td>Read integer from message stream</td>
</tr>
<tr>
<td>readByte</td>
<td>Read byte value from message stream</td>
</tr>
<tr>
<td>readBytes</td>
<td>Read byte array from message stream</td>
</tr>
<tr>
<td>readShort</td>
<td>Read short integer from message stream</td>
</tr>
<tr>
<td>readLong</td>
<td>Read long integer from message stream</td>
</tr>
<tr>
<td>readFloat</td>
<td>Read floating-point value from message stream</td>
</tr>
<tr>
<td>readDouble</td>
<td>Read double-precision value from message stream</td>
</tr>
<tr>
<td>readBoolean</td>
<td>Read boolean value from message stream</td>
</tr>
<tr>
<td>readChar</td>
<td>Read character from message stream</td>
</tr>
<tr>
<td>readString</td>
<td>Read string from message stream</td>
</tr>
<tr>
<td>readObject</td>
<td>Read value from message stream as object</td>
</tr>
</tbody>
</table>

The `readObject` method returns the next value from the message stream in objectified form, as a Java object of the class corresponding to the value's primitive data type: for instance, if the value is of type `int`, `readObject` returns an object of class `Integer`. The following statements are equivalent to the one shown above:

```java
Integer intObject = (Integer) inMsg.readObject();
int intVal = intObject.intValue();
```
Processing Map Messages

The MapMessage interface provides the methods shown in Table 2-22 for reading the body of a map message. Each access method takes a name string as an argument and returns the value to which that name is mapped: for instance, under the example shown in “Composing Map Messages” on page 59, the statement

```java
int meaningOfLife = inMsg.getInt("The Meaning of Life");
```

would set the variable meaningOfLife to the value 42.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getInt</td>
<td>Get integer from message map by name</td>
</tr>
<tr>
<td>getByte</td>
<td>Get byte value from message map by name</td>
</tr>
<tr>
<td>getBytes</td>
<td>Get byte array from message map by name</td>
</tr>
<tr>
<td>getShort</td>
<td>Get short integer from message map by name</td>
</tr>
<tr>
<td>getLong</td>
<td>Get long integer from message map by name</td>
</tr>
<tr>
<td>getFloat</td>
<td>Get floating-point value from message map by name</td>
</tr>
<tr>
<td>getDouble</td>
<td>Get double-precision value from message map by name</td>
</tr>
<tr>
<td>getBoolean</td>
<td>Get boolean value from message map by name</td>
</tr>
<tr>
<td>getChar</td>
<td>Get character from message map by name</td>
</tr>
<tr>
<td>getString</td>
<td>Get string from message map by name</td>
</tr>
<tr>
<td>getObject</td>
<td>Get object from message map by name</td>
</tr>
<tr>
<td>itemExists</td>
<td>Does map contain an item with specified name?</td>
</tr>
<tr>
<td>getMapNames</td>
<td>Get enumeration of all names in map</td>
</tr>
</tbody>
</table>

Like stream messages, map messages provide an access method, `getObject`, that returns a value from the map in objectified form, as a Java object of the class corresponding to the value’s primitive data type: for instance, if the value is of type `int`, `getObject` returns an object of class `Integer`. The following statements are equivalent to the one shown above:

```java
Integer meaningObject = (Integer) inMsg.getObject("The Meaning of Life");
int meaningOfLife = meaningObject.intValue();
```

The `itemExists` method returns a boolean value indicating whether the message map contains an association for a given name string:
if ( inMsg.itemExists("The Meaning of Life") )
{
    /* Life is meaningful */
}
else
{
    /* Life is meaningless */
}

The getMapNames method returns a Java enumeration object for iterating through all of the names defined in the map; you can then use getObject to retrieve the corresponding values, as shown in Example 2–6.

EXAMPLE 2–6  Enumerating Map Message Values

    Enumeration mapNames = inMsg.getMapNames();
    String eachName;
    Object eachValue;
    while ( mapNames.hasMoreElements() )
    {
        eachName = mapNames.nextElement();
        eachValue = inMsg.getObject(eachName);
        /* Do something with the value */
    }

Processing Object Messages

The ObjectMessage interface provides just one method, getObject (Table 2–23), for retrieving the serialized object that is the body of an object message:

    Object messageBody = inMsg.getObject();

You can then typecast the result to a more specific class and process it in whatever way is appropriate.

TABLE 2–23  Object Message Access Method

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getObject</td>
<td>Get serialized object from message body</td>
</tr>
</tbody>
</table>

Processing Bytes Messages

The body of a bytes message simply consists of a stream of uninterpreted bytes; its interpretation is entirely a matter of agreement between sender and receiver. This type of message is intended primarily for decoding message formats used by other existing message systems; Message Queue clients should generally use one of the other, more specific message types instead.
Reading the body of a bytes message is similar to reading a stream message (see "Processing Stream Messages" on page 74); you use the methods shown in Table 2–24 to decode primitive values from the message's byte stream. For example, the statement

```java
t  int intVal = inMsg.readInt();
```

retrieves an integer value from the byte stream. The getBodyLength method returns the length of the entire message body in bytes:

```java
t  int bodyLength = inMsg.getBodyLength();
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getBodyLength</td>
<td>Get length of message body in bytes</td>
</tr>
<tr>
<td>readInt</td>
<td>Read integer from message stream</td>
</tr>
<tr>
<td>readByte</td>
<td>Read signed byte value from message stream</td>
</tr>
<tr>
<td>readUnsignedByte</td>
<td>Read unsigned byte value from message stream</td>
</tr>
<tr>
<td>readBytes</td>
<td>Read byte array from message stream</td>
</tr>
<tr>
<td>readShort</td>
<td>Read signed short integer from message stream</td>
</tr>
<tr>
<td>readUnsignedShort</td>
<td>Read unsigned short integer from message stream</td>
</tr>
<tr>
<td>readLong</td>
<td>Read long integer from message stream</td>
</tr>
<tr>
<td>readFloat</td>
<td>Read floating-point value from message stream</td>
</tr>
<tr>
<td>readDouble</td>
<td>Read double-precision value from message stream</td>
</tr>
<tr>
<td>readBoolean</td>
<td>Read boolean value from message stream</td>
</tr>
<tr>
<td>readChar</td>
<td>Read character from message stream</td>
</tr>
<tr>
<td>readUTF</td>
<td>Read UTF-8 string from message stream</td>
</tr>
</tbody>
</table>
Message Queue Clients: Design and Features

This chapter addresses architectural and configuration issues that depend upon Message Queue's implementation of the Java Message Specification. It covers the following topics:

- “Client Design Considerations” on page 79
- “Managing Client Threads” on page 84
- “Managing Memory and Resources” on page 86
- “Programming Issues for Message Consumers” on page 92
- “Factors Affecting Performance” on page 94
- “Connection Event Notification” on page 98
- “Client Connection Failover (Auto-Reconnect)” on page 100
- “Custom Client Acknowledgment” on page 135
- “Communicating with C.Clients” on page 139
- “Client Runtime Logging” on page 139

Client Design Considerations

The choices you make in designing a JMS client affect portability, allocating work between connections and sessions, reliability and performance, resource use, and ease of administration. This section discusses basic issues that you need to address in client design. It covers the following topics:

- “Developing Portable Clients” on page 80
- “Choosing Messaging Domains” on page 80
- “Connections and Sessions” on page 81
- “Producers and Consumers” on page 82
- “Balancing Reliability and Performance” on page 84
Developing Portable Clients

The Java Messaging Specification was developed to abstract access to message-oriented middleware systems (MOMs). A client that writes JMS code should be portable to any provider that implements this specification. If code portability is important to you, be sure that you do the following in developing clients:

■ Make sure your code does not depend on extensions or features that are specific to Message Queue.
■ Look up, using JNDI, (rather than instantiate) administered objects for connection factories and destinations.

Administered objects encapsulate provider-specific implementation and configuration information. Besides allowing for portability, administered objects also make it much easier to share connection factories between applications and to tune a JMS application for performance and resource use. So, even if portability is not important to you, you might still want to leave the work of creating and configuring these objects to an administrator. For more information, see “Looking Up a Connection Factory With JNDI” on page 35 and “Looking Up a Destination With JNDI” on page 43.

Choosing Messaging Domains

As described in “Messaging Domains” in Oracle GlassFish Message Queue 4.4.2 Technical Overview, JMS supports two distinct message delivery models: point-to-point and publish/subscribe. These two message delivery models can be handled using different API objects—with slightly different semantics—representing different programming domains, as shown in Table 3–1, or they can be handled by base (unified domain) types.

<table>
<thead>
<tr>
<th>Unified Domain (Queue or Topic)</th>
<th>Point-to-Point Domain</th>
<th>Publish/Subscribe Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destination</td>
<td>Queue</td>
<td>Topic</td>
</tr>
<tr>
<td>ConnectionFactory</td>
<td>QueueConnectionFactory</td>
<td>TopicConnectionFactory</td>
</tr>
<tr>
<td>Connection</td>
<td>QueueConnection</td>
<td>TopicConnection</td>
</tr>
<tr>
<td>Session</td>
<td>QueueSession</td>
<td>TopicSession</td>
</tr>
<tr>
<td>MessageProducer</td>
<td>QueueSender</td>
<td>TopicPublisher</td>
</tr>
<tr>
<td>MessageConsumer</td>
<td>QueueReceiver</td>
<td>TopicSubscriber</td>
</tr>
</tbody>
</table>

Using the point-to-point or publish/subscribe domains offers the advantage of a clean API that prevents certain types of programming errors; for example, creating a durable subscriber for a queue destination. However, the non-unified domains have the disadvantage that you cannot
combine point-to-point and publish/subscribe operations in the same transaction or in the same session. If you need to do that, you should choose the unified domain API.

The JMS 1.1 specification continues to support the separate JMS 1.0.2 programming domains. (The example applications included with the Message Queue product as well as the code examples provided in this book all use the separate JMS 1.0.2 programming domains.) You can choose the API that best suits your needs. The only exception are those developers needing to write clients for the Sun Java System Application Server 7 environment, as explained in the following note.

**Note** – Developers of applications that run in the Sun Java System Application Server 7 environment are limited to using the JMS 1.0.2 API. This is because Sun Java System Application Server 7 complies with the J2EE 1.3 specification, which supports only JMS 1.0.2. Any JMS messaging performed in servlets and EJBs—including message-driven beans must be based on the domain-specific JMS APIs and cannot use the JMS 1.1 unified domain APIs. Developers of J2EE applications that will run in J2EE 1.4-compliant servers can, however, use the simpler JMS 1.1 APIs.

## Connections and Sessions

A connection is a relatively heavy-weight object because of the authentication and communication setup that must be done when a connection is created. For this reason, it’s a good idea to use as few connections as possible. The real allocation of work occurs in sessions, which are light-weight, single-threaded contexts for producing and consuming messages. When you are thinking about structuring your client, it is best to think of the work that is done at the session level.

A session

- Is a factory for its message producers and consumers
- Supplies provider-optimized message factories
- Supports a single series of transactions that combine work spanning its producers and consumers into atomic units
- Defines a serial order for the messages it consumes and the messages it produces
- Retains messages until they have been acknowledged
- Serializes execution of message listeners registered with its message consumers

The requirement that sessions be operated on by a single thread at a time places some restrictions on the combination of producers and consumers that can use the same session. In particular, if a session has an asynchronous consumer, it may not have any other synchronous consumers. For a discussion of the connection and session's use of threads, see "Managing Client Threads" on page 84. With the exception of these restrictions, let the needs of your application determine the number of sessions, producers, and consumers.
Producers and Consumers

Aside from the reliability your client requires, the design decisions that relate to producers and consumers include the following:

- Do you want to use a point-to-point or a publish/subscribe domain?
  There are some interesting permutations here. There are times when you would want to use publish/subscribe even when you have only one subscriber. On the other hand, performance considerations might make the point-to-point model more efficient than the publish/subscribe model, when the work of sorting messages between subscribers is too costly. Sometimes you cannot make these decisions in the abstract, but must actually develop and test different prototypes.

- Are you using an asynchronous message consumer that does not receive messages often or a producer that is seldom used?
  Let the administrator know how to set the ping interval, so that your client gets an exception if the connection should fail. For more information see "Using the Client Runtime Ping Feature" on page 92.

- Are you using a synchronous consumer in a distributed application?
  You might need to allow a small time interval between connecting and calling the `receiveNoWait()` method in order not to miss a pending message. For more information, see "Synchronous Consumption in Distributed Applications" on page 93.

- Do you need message compression?
  Benefits vary with the size and format of messages, the number of consumers, network bandwidth, and CPU performance; and benefits are not guaranteed. For a more detailed discussion, see "Message Compression" on page 87.

Assigning Client Identifiers

A connection can have a client identifier. This identifier is used to associate a JMS client’s connection to a message service, with state information maintained by the message service for that client. The JMS provider must ensure that a client identifier is unique, and applies to only one connection at a time. Currently, client identifiers are used to maintain state for durable subscribers. In defining a client identifier, you can use a special variable substitution syntax that allows multiple connections to be created from a single `ConnectionFactory` object using different username parameters to generate unique client identifiers. These connections can be used by multiple durable subscribers without naming conflicts or lack of security.

Message Queue allows client identifiers to be set in one of two ways:

- **Programmatically**: You use the `setClientID` method of the `Connection` object. If you use this method, you must set the client id before you use the connection. Once the connection is used, the client identifier cannot be set or reset.
Administratively: The administrator specifies the client ID when creating the connection factory administrative object. See "Client Identifier" in Oracle GlassFish Message Queue 4.4.2 Administration Guide.

Message Order and Priority

In general, all messages sent to a destination by a single session are guaranteed to be delivered to a consumer in the order they were sent. However, if they are assigned different priorities, a messaging system will attempt to deliver higher priority messages first.

Beyond this, the ordering of messages consumed by a client can have only a rough relationship to the order in which they were produced. This is because the delivery of messages to a number of destinations and the delivery from those destinations can depend on a number of issues that affect timing, such as the order in which the messages are sent, the sessions from which they are sent, whether the messages are persistent, the lifetime of the messages, the priority of the messages, the message delivery policy of queue destinations (see Chapter 18, “Physical Destination Property Reference,” in Oracle GlassFish Message Queue 4.4.2 Administration Guide), and message service availability.

Using Selectors Efficiently

The use of selectors can have a significant impact on the performance of your application. It’s difficult to put an exact cost on the expense of using selectors since it varies with the complexity of the selector expression, but the more you can do to eliminate or simplify selectors the better.

One way to eliminate (or simplify) selectors is to use multiple destinations to sort messages. This has the additional benefit of spreading the message load over more than one producer, which can improve the scalability of your application. For those cases when it is not possible to do that, here are some techniques that you can use to improve the performance of your application when using selectors:

■ Have consumers share selectors. As of version 3.5 of Message Queue, message consumers with identical selectors “share” that selector in imqbrokerd which can significantly improve performance. So if there is a way to structure your application to have some selector sharing, consider doing so.

■ Use IN instead of multiple string comparisons. For example, the following expression:

        color IN ('red', 'green', 'white')

is much more efficient than this expression

        color = 'red' OR color = 'green' OR color = 'white'

especially if the above expression usually evaluates to false.

■ Use BETWEEN instead of multiple integer comparisons. For example:

        size BETWEEN 6 AND 10
is generally more efficient than

\[ \text{size} \geq 6 \text{ AND } \text{size} \leq 10 \]

especially if the above expression usually evaluates to true.

- Order the selector expression so that Message Queue can determine its evaluation as soon as possible. (Evaluation proceeds from left to right.) This can easily double or triple performance when using selectors, depending on the complexity of the expression.

- If you have two expressions joined by an OR, put the expression that is most likely to evaluate to TRUE first.

- If you have two expressions joined by an AND, put the expression that is most likely to evaluate to FALSE first.

For example, if `size` is usually greater than 6, but `color` is rarely red you'd want the order of an OR expression to be:

\[ \text{size} > 6 \text{ OR } \text{color} = 'red' \]

If you are using AND:

\[ \text{color} = 'red' \text{ AND } \text{size} > 6 \]

**Balancing Reliability and Performance**

Reliable messaging is implemented in a variety of ways: through the use of persistent messages, acknowledgments or transactions, durable subscriptions, and connection failover.

In general, the more reliable the delivery of messages, the more overhead and bandwidth are required to achieve it. The trade-off between reliability and performance is a significant design consideration. You can maximize *performance* and throughput by choosing to produce and consume nonpersistent messages. On the other hand, you can maximize *reliability* by producing and consuming persistent messages in a transaction using a transacted session. For a detailed discussion of design options and their impact on performance, see "Factors Affecting Performance" on page 94.

**Managing Client Threads**

Using client threads effectively requires that you balance performance, throughput, and resource needs. To do this, you need to understand JMS restrictions on thread usage, what threads Message Queue allocates for itself, and the architecture of your applications. This section addresses these issues and offers some guidelines for managing client threads.
JMS Threading Restrictions

The Java Messaging Specification mandates that a session not be operated on by more than one thread at a time. This leads to the following restrictions:

- A session may not have an asynchronous consumer and a synchronous consumer.
- A session that has an asynchronous consumer can only produce messages from within the `onMessage()` method (the message listener). The only call that you can make outside the message listener is to close the session.
- A session may include any number of synchronous consumers, any number of producers, and any combination of the two. That is, the single-thread requirement cannot be violated by these combinations. However, performance may suffer.

The system does not enforce the requirement that a session be single threaded. If your client application violates this requirement, you will get a `JMSIllegalStateException` exception or unexpected results.

Thread Allocation for Connections

When the Message Queue client runtime creates a connection, it creates two threads: one for consuming messages from the socket, and one to manage the flow of messages for the connection. In addition, the client runtime creates a thread for each client session. Thus, at a minimum, for a connection using one session, three threads are created. For a connection using three sessions, five threads are created, and so on.

Managing threads in a JMS application often involves trade-offs between performance and throughput. Weigh the following considerations when dealing with threading issues.

- When you create several asynchronous message consumers in the same session, messages are delivered serially by the session thread to these consumers. Sharing a session among several message consumers might starve some consumers of messages while inundating other consumers. If the message rate across these consumers is high enough to cause an imbalance, you might want to separate the consumers into different sessions. To determine whether message flow is unbalanced, you can monitor destinations to see the rate of messages coming in. See Chapter 4, “Using the Metrics Monitoring API.”
- You can reduce the number of threads allocated to the client application by using fewer connections and fewer sessions. However, doing this might slow your application’s throughput.
- You might be able to use certain JVM runtime options to improve thread memory usage and performance. For example, if you are running on the Solaris platform, you may be able to run with the same number (or more) threads by using the following `vm` options with the client: Refer to the JDK documentation for details.
  - Use the `Xss128K` option to decrease the memory size of the heap.
Use the `xconcurrentIO` option to improve thread performance in the 1.3 VM.

## Managing Memory and Resources

This section describes memory and performance issues that you can manage by increasing JVM heap space and by managing the size of your messages. It covers the following topics:

- “Managing Memory” on page 86
- “Managing Message Size” on page 86
- “Managing the Dead Message Queue” on page 88
- “Managing Physical Destination Limits” on page 92

You can also improve performance by having the administrator set connection factory attributes to meter the message flow over the client-broker connection and to limit the message flow for a consumer. For a detailed explanation, please see "Reliability And Flow Control" in *Oracle GlassFish Message Queue 4.4.2 Administration Guide*.

### Managing Memory

A client application running in a JVM needs enough memory to accommodate messages that flow in from the network as well as messages the client creates. If your client gets `OutOfMemoryError` errors, chances are that not enough memory was provided to handle the size or the number of messages being consumed or produced.

Your client might need more than the default JVM heap space. On most systems, the default is 64 MB but you will need to check the default values for your system.

Consider the following guidelines:

- Evaluate the normal and peak system memory footprints when sizing heap space.
- You can start by doubling the heap size using a command like the following:
  ```java
  java -Xmx128m MyClass
  ```
- The best size for the heap space depends on both the operating system and the JDK release. Check the JDK documentation for restrictions.
- The size of the VM’s memory allocation pool must be less than or equal to the amount of virtual memory that is available on the system.

### Managing Message Size

In general, for better manageability, you can break large messages into smaller parts, and use sequencing to ensure that the partial messages sent are concatenated properly. You can also use
a Message Queue JMS feature to compress the body of a message. This section describes the
programming interface that allows you to compress messages and to compare the size of
compressed and uncompressed messages.

Message compression and decompression is handled entirely by the client runtime, without
involving the broker. Therefore, applications can use this feature with a previous version of the
broker, but they must use version 3.6 or later of the Message Queue client runtime library.

**Message Compression**

You can use the `Message.setBooleanProperty()` method to specify that the body of a message
be compressed. If the `JMS_SUN_COMPRESS` property is set to `true`, the client runtime, will
compress the body of the message being sent. This happens after the producer's send method is
called and before the send method returns to the caller. The compressed message is
automatically decompressed by the client runtime before the message is delivered to the
message consumer.

For example, the following call specifies that a message be compressed:

```java
MyMessage.setBooleanProperty("JMS_SUN_COMPRESS",true);
```

Compression only affects the message body; the message header and properties are not
compressed.

Two read-only JMS message properties are set by the client runtime after a message is sent.
Applications can test the properties (JMS_SUN_UNCOMPRESSED_SIZE and
JMS_SUN_COMPRESSED_SIZE) after a send returns to determine whether compression is
advantageous. That is, applications wanting to use this feature, do not have to explicitly receive
a compressed and uncompressed version of the message to determine whether compression is
desired.

If the consumer of a compressed message wants to resend the message in an uncompressed
form, it should call the `Message.clearProperties()` to clear the `JMS_SUN_COMPRESS` property.
Otherwise, the message will be compressed before it is sent to its next destination.

**Advantages and Limitations of Compression**

Although message compression has been added to improve performance, such benefit is not
guaranteed. Benefits vary with the size and format of messages, the number of consumers,
network bandwidth, and CPU performance. For example, the cost of compression and
decompression might be higher than the time saved in sending and receiving a compressed
message. This is especially true when sending small messages in a high-speed network. On the
other hand, applications that publish large messages to many consumers or who publish in a
slow network environment, might improve system performance by compressing messages.
Depending on the message body type, compression may also provide minimal or no benefit. An application client can use the `JMS_SUN_UNCOMPRESSED_SIZE` and `JMS_SUN_COMPRESSED_SIZE` properties to determine the benefit of compression for different message types.

Message consumers deployed with client runtime libraries that precede version 3.6 cannot handle compressed messages. Clients wishing to send compressed messages must make sure that consumers are compatible. C clients cannot currently consume compressed messages.

**Compression Examples**

**Example 3–1** shows how you set and send a compressed message:

**EXAMPLE 3–1  Sending a Compressed Message**

```java
//topicSession and myTopic are assumed to have been created
topicPublisher publisher = topicSession.createPublisher(myTopic);
BytesMessage bytesMessage = topicSession.createBytesMessage();

//byteArray is assumed to have been created
bytesMessage.writeBytes(byteArray);

//instruct the client runtime to compress this message
bytesMessage.setBooleanProperty("JMS_SUN_COMPRESS", true);

//publish message to the myTopic destination
publisher.publish(bytesMessage);
```

**Example 3–2** shows how you examine compressed and uncompressed message body size. The `bytesMessage` was created as in **Example 3–1**:

**EXAMPLE 3–2  Comparing Compressed and Uncompressed Message Size**

```java
//get uncompressed body size
int uncompressed = bytesMessage.getIntProperty("JMS_SUN_UNCOMPRESSED_SIZE");

//get compressed body size
int compressed = bytesMessage.getIntProperty("JMS_SUN_COMPRESSED_SIZE");
```

**Managing the Dead Message Queue**

When a message is deemed undeliverable, it is automatically placed on a special queue called the dead message queue. A message placed on this queue retains all of its original headers (including its original destination) and information is added to the message’s properties to explain why it became a dead message. An administrator or a developer can access this queue, remove a message, and determine why it was placed on the queue.
For an introduction to dead messages and the dead message queue, see "Using the Dead Message Queue" in *Oracle GlassFish Message Queue 4.4.2 Administration Guide*.

For a description of the destination properties and of the broker properties that control the system’s use of the dead message queue, see Chapter 18, “Physical Destination Property Reference,” in *Oracle GlassFish Message Queue 4.4.2 Administration Guide*.

This section describes the message properties that you can set or examine programmatically to determine the following:

- Whether a dead message can be sent to the dead message queue.
- Whether the broker should log information when a message is destroyed or moved to the dead message queue.
- Whether the body of the message should also be stored when the message is placed on the dead message queue.
- Why the message was placed on the dead message queue and any ancillary information.

Message Queue 3.6 clients can set properties related to the dead message queue on messages and send those messages to clients compiled against earlier versions. However clients receiving such messages cannot examine these properties without recompiling against 3.6 libraries.

The dead message queue is automatically created by the broker and called `mq.sys.dmq`. You can use the message monitoring API, described in Chapter 4, "Using the Metrics Monitoring API," to determine whether that queue is growing, to examine messages on that queue, and so on.

You can set the properties described in Table 3–2 for any message to control how the broker should handle that message if it deems it to be undeliverable. Note that these message properties are needed only to override destination, or broker-based behavior.

**TABLE 3–2  Message Properties Relating to Dead Message Queue**

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS_SUN_PRESERVE_UNDELIVERED</td>
<td>A boolean whose value determines what the broker should do with the message if it is dead. The default value of unset, specifies that the message should be handled as specified by the useDMQ property of the destination to which the message was sent. A value of true overrides the setting of the useDMQ property and sends the dead message to the dead message queue. A value of false overrides the setting of the useDMQ property and prevents the dead message from being placed in the dead message queue.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>JMS_SUN_LOG_DEAD_MESSAGES</td>
<td>A boolean value that determines how activity relating to dead messages should be logged. The default value of unset, will behave as specified by the broker configuration property imq.destination.logDeadMsgs. A value of true overrides the setting of the imq.destination.logDeadMsgs broker property and specifies that the broker should log the action of removing a message or moving it to the dead message queue. A value of false overrides the setting of the imq.destination.logDeadMsgs broker property and specifies that the broker should not log these actions.</td>
</tr>
<tr>
<td>JMS_SUN_TRUNCATE_MSG_BODY</td>
<td>A boolean value that specifies whether the body of a dead message is truncated. The default value of unset, will behave as specified by the broker property imq.destination.DMQ.truncateBody. A value of true overrides the setting of the imq.destination.DMQ.truncateBody property and specifies that the body of the message should be discarded when the message is placed in the dead message queue. A value of false overrides the setting of the imq.destination.DMQ.truncateBody property and specifies that the body of the message should be stored along with the message header and properties when the message is placed in the dead message queue.</td>
</tr>
</tbody>
</table>

The properties described in Table 3–3 are set by the broker for a message placed in the dead message queue. You can examine the properties for the message to retrieve information about why the message was placed on the queue and to gather other information about the message and about the context within which this action was taken.
### TABLE 3-3 Dead Message Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSXDeliveryCount</td>
<td>An Integer that specifies the most number of times the message was delivered to a given consumer. This value is set only for ERROR or UNDELIVERABLE messages.</td>
</tr>
<tr>
<td>JMS_SUN_DMQ_UNDELIVERED_TIMESTAMP</td>
<td>A Long that specifies the time (in milliseconds) when the message was placed on the dead message queue.</td>
</tr>
<tr>
<td>JMS_SUN_DMQ_UNDELIVERED_REASON</td>
<td>A string that specifies one of the following values to indicate the reason why the message was placed on the dead message queue: OLDEST, LOW_PRIORITY, EXPIRED, UNDELIVERABLE, ERROR. If the message was marked dead for multiple reasons, for example it was undeliverable and expired, only one reason will be specified by this property. The ERROR reason indicates that an internal error made it impossible to process the message. This is an extremely unusual condition, and the sender should just resend the message.</td>
</tr>
<tr>
<td>JMS_SUN_DMQ_PRODUCING_BROKER</td>
<td>A String used for message traffic in broker clusters: it specifies the broker name and port number of the broker that produced the message. A null value indicates the local broker.</td>
</tr>
<tr>
<td>JMS_SUN_DMQ_DEAD_BROKER</td>
<td>A String used for message traffic in broker clusters: it specifies the broker name and port number of the broker that placed the message on the dead message queue. A null value indicates the local broker.</td>
</tr>
<tr>
<td>JMS_SUN_DMQ_UNDELIVERED_EXCEPTION</td>
<td>A String that specifies the name of the exception (if the message was dead because of an exception) on either the client or the broker.</td>
</tr>
<tr>
<td>JMS_SUN_DMQ_UNDELIVERED_COMMENT</td>
<td>A String used to provide an optional comment when the message is marked dead.</td>
</tr>
</tbody>
</table>
TABLE 3–3  Dead Message Properties  (Continued)

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS_SUN_DMQ_BODY_TRUNCATED</td>
<td>A Boolean: a value of true indicates that the message body was not stored. A value of false indicates that the message body was stored.</td>
</tr>
</tbody>
</table>

**Managing Physical Destination Limits**

When creating a topic or queue destination, the administrator can specify how the broker should behave when certain memory limits are reached. Specifically, when the number of messages reaching a physical destination exceeds the number specified with the `maxNumMsgs` property or when the total amount of memory allowed for messages exceeds the number specified with the `maxTotalMsgBytes` property, the broker takes one of the following actions, depending on the setting of the `limitBehavior` property:

- Slows message producers (`FLOW\_CONTROL`)
- Throws out the oldest message in memory (`REMOVE\_OLDEST`)
- Throws out the lowest priority message in memory (`REMOVE\_LOW\_PRIORITY`)
- Rejects the newest messages (`REJECT\_NEWEST`)

If the default value `REJECT\_NEWEST` is specified for the `limitBehavior` property, the broker throws out the newest messages received when memory limits are exceeded. If the message discarded is a persistent message, the producing client gets an exception which should be handled by resending the message later.

If any of the other values is selected for the `limitBehavior` property or if the message is not persistent, the application client is not notified if a message is discarded. Application clients should let the administrator know how they prefer this property to be set for best performance and reliability.

**Programming Issues for Message Consumers**

This section describes two problems that consumers might need to manage: the undetected loss of a connection, or the loss of a message for distributed synchronous consumers.

**Using the Client Runtime Ping Feature**

Message Queue defines a connection factory attribute for a ping interval. This attribute specifies the interval at which the client runtime should check the client’s connection to the broker. The ping feature is especially useful to Message Queue clients that exclusively receive messages and might therefore not be aware that the absence of messages is due to a connection failure. This feature could also be useful to producers who don’t send messages frequently and who would want notification that a connection they’re planning to use is not available.
The connection factory attribute used to specify this interval is called `imqPingInterval`. Its default value is 30 seconds. A value of -1 or 0, specifies that the client runtime should not check the client connection.

Developers should set (or have the administrator set) ping intervals that are slightly more frequent than they need to send or receive messages, to allow time to recover the connection in case the ping discovers a connection failure. Note also that the ping may not occur at the exact time specified by the value you supply for `interval`; the underlying operating system’s use of i/o buffers may affect the amount of time needed to detect a connection failure and trigger an exception.

A failed ping operation results in a `JMSException` on the subsequent method call that uses the connection. If an exception listener is registered on the connection, it will be called when a ping operation fails.

### Preventing Message Loss for Synchronous Consumers

It is always possible that a message can be lost for synchronous consumers in a session using `AUTO_ACKNOWLEDGE` mode if the provider fails. To prevent this possibility, you should either use a transacted session or a session in `CLIENT_ACKNOWLEDGE` mode.

### Synchronous Consumption in Distributed Applications

Because distributed applications involve greater processing time, such an application might not behave as expected if it were run locally. For example, calling the `receiveNoWait` method for a synchronous consumer might return `null` even when there is a message available to be retrieved.

If a client connects to the broker and immediately calls the `receiveNoWait` method, it is possible that the message queued for the consuming client is in the process of being transmitted from the broker to the client. The client runtime has no knowledge of what is on the broker, so when it sees that there is no message available on the client’s internal queue, it returns with a `null`, indicating no message.

You can avoid this problem by having your client do either of the following:

- Use one of the synchronous receive methods that specifies a timeout interval.
- Use a queue browser to check the queue before calling the `receiveNoWait` method.
Factors Affecting Performance

Application design decisions can have a significant effect on overall messaging performance. The most important factors affecting performance are those that impact the reliability of message delivery; among these are the following:

- “Delivery Mode (Persistent/Nonpersistent)” on page 95
- “Use of Transactions” on page 95
- “Acknowledgment Mode” on page 95
- “Durable vs. Nondurable Subscriptions” on page 96

Other application design factors impacting performance include the following:

- “Use of Selectors (Message Filtering)” on page 97
- “Message Size” on page 97
- “Message Body Type” on page 98

The sections that follow describe the impact of each of these factors on messaging performance. As a general rule, there is a trade-off between performance and reliability: factors that increase reliability tend to decrease performance.

Table 3–4 shows how application design factors affect messaging performance. The table shows two scenarios—a high-reliability, low-performance scenario and a high-performance, low-reliability scenario—and the choice of application design factors that characterizes each. Between these extremes, there are many choices and trade-offs that affect both reliability and performance.

<table>
<thead>
<tr>
<th>Application Design Factor</th>
<th>High Reliability, Low Performance</th>
<th>High Performance, Low Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delivery mode</td>
<td>Persistent messages</td>
<td>Nonpersistent messages</td>
</tr>
<tr>
<td>Use of transactions</td>
<td>Transacted sessions</td>
<td>No transactions</td>
</tr>
<tr>
<td>Acknowledgment mode</td>
<td>AUTO_ACKNOWLEDGE</td>
<td>DUPS_OK_ACKNOWLEDGE</td>
</tr>
<tr>
<td></td>
<td>CLIENT_ACKNOWLEDGE</td>
<td>NO_ACKNOWLEDGE</td>
</tr>
<tr>
<td>Durable/nondurable subscriptions</td>
<td>Durable subscriptions</td>
<td>Nondurable subscriptions</td>
</tr>
<tr>
<td>Use of selectors</td>
<td>Message filtering</td>
<td>No message filtering</td>
</tr>
<tr>
<td>Message size</td>
<td>Small messages</td>
<td>Large messages</td>
</tr>
<tr>
<td>Message body type</td>
<td>Complex body types</td>
<td>Simple body types</td>
</tr>
</tbody>
</table>
Delivery Mode (Persistent/Nonpersistent)

Persistent messages guarantee message delivery in case of broker failure. The broker stores these messages in a persistent store until all intended consumers acknowledge that they have consumed the message.

Broker processing of persistent messages is slower than for nonpersistent messages for the following reasons:

- A broker must reliably store a persistent message so that it will not be lost should the broker fail.
- The broker must confirm receipt of each persistent message it receives. Delivery to the broker is guaranteed once the method producing the message returns without an exception.
- Depending on the client acknowledgment mode, the broker might need to confirm a consuming client’s acknowledgment of a persistent message.

For both queues and topics with durable subscribers, performance was approximately 40% faster for non-persistent messages. We obtained these results using 10K-size messages and AUTO_Acknowledge mode.

Use of Transactions

A transaction guarantees that all messages produced in a transacted session and all messages consumed in a transacted session will be either processed or not processed (rolled back) as a unit. Message Queue supports both local and distributed transactions.

A message produced or acknowledged in a transacted session is slower than in a non-transacted session for the following reasons:

- Additional information must be stored with each produced message.
- In some situations, messages in a transaction are stored when normally they would not be. For example, a persistent message delivered to a topic destination with no subscriptions would normally be deleted, however, at the time the transaction is begun, information about subscriptions is not available.
- Information on the consumption and acknowledgment of messages within a transaction must be stored and processed when the transaction is committed.

Acknowledgment Mode

Other than using transactions, you can ensure reliable delivery by having the client acknowledge receiving a message. If a session is closed without the client acknowledging the message or if the message broker fails before the acknowledgment is processed, the broker redelivers that message, setting a JMSRedelivered flag.
For a non-transacted session, the client can choose one of three acknowledgment modes, each of which has its own performance characteristics:

- **AUTO_ACKNOWLEDGE.** The system automatically acknowledges a message once the consumer has processed it. This mode guarantees at most one redelivered message after a provider failure.

- **CLIENT_ACKNOWLEDGE.** The application controls the point at which messages are acknowledged. All messages processed in that session since the previous acknowledgment are acknowledged. If the broker fails while processing a set of acknowledgments, one or more messages in that group might be redelivered.

  (Using CLIENT_ACKNOWLEDGE mode is similar to using transactions, except there is no guarantee that all acknowledgments will be processed together if a provider fails during processing.)

- **DUPS_OK_ACKNOWLEDGE.** This mode instructs the system to acknowledge messages in a lazy manner. Multiple messages can be redelivered after a provider failure.

- **NO_ACKNOWLEDGE** In this mode, the broker considers a message acknowledged as soon as it has been written to the client. The broker does not wait for an acknowledgment from the receiving client. This mode is best used by typic subscribers who are not worried about reliability.

Performance is impacted by acknowledgment mode for the following reasons:

- Extra control messages between broker and client are required in AUTO_ACKNOWLEDGE and CLIENT_ACKNOWLEDGE modes. The additional control messages add processing overhead and can interfere with JMS payload messages, causing processing delays.

- In AUTO_ACKNOWLEDGE and CLIENT_ACKNOWLEDGE modes, the client must wait until the broker confirms that it has processed the client’s acknowledgment before the client can consume more messages. (This broker confirmation guarantees that the broker will not inadvertently redeliver these messages.)

- The Message Queue persistent store must be updated with the acknowledgment information for all persistent messages received by consumers, thereby decreasing performance.

### Durable vs. Nondurable Subscriptions

Subscribers to a topic destination have either durable and nondurable subscriptions. Durable subscriptions provide increased reliability at the cost of slower throughput for the following reasons:

- The Message Queue message broker must persistently store the list of messages assigned to each durable subscription so that should the broker fail, the list is available after recovery.
Persistent messages for durable subscriptions are stored persistently, so that should a broker fail, the messages can still be delivered after recovery, when the corresponding consumer becomes active. By contrast, persistent messages for nondurable subscriptions are not stored persistently (should a broker fail, the corresponding consumer connection is lost and the message would never be delivered).

We compared performance for durable and non-durable subscribers in two cases: persistent and nonpersistent 10k-sized messages. Both cases use AUTO_ACKNOWLEDGE acknowledgment mode. We found a performance impact only in the case of persistent messages, which slowed messages conveyed to durable subscribers by about 30%.

Use of Selectors (Message Filtering)

Application developers can have the messaging provider sort messages according to criteria specified in the message selector associated with a consumer and deliver to that consumer only those messages whose property value matches the message selector. For example, if an application creates a subscriber to the topic WidgetOrders and specifies the expression NumberOfOrders > 1000 for the message selector, messages with a NumberOfOrders property value of 1001 or more are delivered to that subscriber.

Creating consumers with selectors lowers performance (as compared to using multiple destinations) because additional processing is required to handle each message. When a selector is used, it must be parsed so that it can be matched against future messages. Additionally, the message properties of each message must be retrieved and compared against the selector as each message is routed. However, using selectors provides more flexibility in a messaging application and may lower resource requirements at the expense of speed.

Message Size

Message size affects performance because more data must be passed from producing client to broker and from broker to consuming client, and because for persistent messages a larger message must be stored.

However, by batching smaller messages into a single message, the routing and processing of individual messages can be minimized, providing an overall performance gain. In this case, information about the state of individual messages is lost.

In our tests, which compared throughput in kilobytes per second for 1K, 10K, and 100K-sized messages to a queue destination using AUTO_ACKNOWLEDGE mode, we found that non-persistent messaging was about 50% faster for 1K messages, about 20% faster for 10K messages, and about 5% faster for 100K messages. The size of the message affected performance significantly for both persistent and non-persistent messages. 100k messages are about 10 times faster than 10K, and 10K messages are about 5 times faster than 1K.
Message Body Type

JMS supports five message body types, shown below roughly in the order of complexity:

- **Bytes**: Contains a set of bytes in a format determined by the application
- **Text**: Is a simple `java.lang.String`
- **Stream**: Contains a stream of Java primitive values
- **Map**: Contains a set of name-and-value pairs
- **Object**: Contains a Java serialized object

While, in general, the message type is dictated by the needs of an application, the more complicated types (map and object) carry a performance cost — the expense of serializing and deserializing the data. The performance cost depends on how simple or how complicated the data is.

Connection Event Notification

Connection event notifications allow a Message Queue client to listen for closure and reconnection events and to take appropriate action based on the notification type and the connection state. For example, when a failover occurs and the client is reconnected to another broker, an application might want to clean up its transaction state and proceed with a new transaction.

If the Message Queue provider detects a serious problem with a connection, it calls the connection object's registered exception listener. It does this by calling the listener's `onException` method, and passing it a `JMSException` argument describing the problem. In the same way, the Message Queue provider offers an event notification API that allows the client runtime to inform the application about connection state changes. The notification API is defined by the following elements:

- The `com.sun.messaging.jms.notification` package, which defines the event listener and the notification event objects.
- The `com.sun.messaging.jms.Connection` interface, which defines extensions to the `javax.jms.Connection` interface.

The following sections describe the events that can trigger notification and explain how you can create an event listener.

Connection Events

The following table lists and describes the events that can be returned by the event listener.

Note that the JMS exception listener is not called when a connection event occurs. The exception listener is only called if the client runtime has exhausted its reconnection attempts. The client runtime always calls the event listener before the exception listener.
### TABLE 3-5 Notification Events

<table>
<thead>
<tr>
<th>Event Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConnectionClosingEvent</td>
<td>The Message Queue client runtime generates this event when it receives a notification from the broker that a connection is about to be closed due to a shutdown requested by the administrator.</td>
</tr>
<tr>
<td>ConnectionClosedEvent</td>
<td>The Message Queue client runtime generates this event when a connection is closed due to a broker error or when it is closed due to a shutdown or restart requested by the administrator. When an event listener receives a ConnectionClosedEvent, the application can use the <code>getEventCode()</code> method of the received event to get an event code that specifies the cause for closure.</td>
</tr>
<tr>
<td>ConnectionReconnectedEvent</td>
<td>The Message Queue client runtime has reconnected to a broker. This could be the same broker to which the client was previously connected or a different broker. An application can use the <code>getBrokerAddress</code> method of the received event to get the address of the broker to which it has been reconnected.</td>
</tr>
<tr>
<td>ConnectionReconnectFailedEvent</td>
<td>The Message Queue client runtime has failed to reconnect to a broker. Each time a reconnect attempt fails, the runtime generates a new event and delivers it to the event listener. The JMS exception listener is not called when a connection event occurs. It is only called if the client runtime has exhausted its reconnection attempts. The client runtime always calls the event listener before the exception listener.</td>
</tr>
</tbody>
</table>

## Creating an Event Listener

The following code example illustrates how you set a connection event listener. Whenever a connection event occurs, the event listener's `onEvent` method will be invoked by the client runtime.

```java
//create an MQ connection factory.
com.sun.messaging.ConnectionFactory factory =
    new com.sun.messaging.ConnectionFactory();

//create an MQ connection.
```
com.sun.messaging.jms.Connection connection =
    (com.sun.messaging.jms.Connection )factory.createConnection();

//construct an MQ event listener. The listener implements
com.sun.messaging.jms.notification.EventListener eListener =
    new ApplicationEventListener();

//set event listener to the MQ connection.
connection.setEventListener ( eListener );

EventListenerExamples
In this example, an application chooses to have its event listener log the connection event to the
application's logging system.

public class ApplicationEventListener implements
    com.sun.messaging.jms.notification.EventListener {

    public void onEvent ( com.sun.messaging.jms.notification.Event connEvent ) {
        log (connEvent);
    }

    private void log ( com.sun.messaging.jms.notification.Event connEvent ) {
        String eventCode = connEvent.getEventCode();
        String eventMessage = connEvent.getEventMessage();
        //write event information to the output stream.
    }
}

Client Connection Failover (Auto-Reconnect)
Message Queue supports client connection failover. A failed connection can be automatically
restored not only to the original broker, but to a different broker in a broker cluster. There are
circumstances under which the client-side state cannot be restored on any broker during an
automatic reconnection attempt; for example, when the client uses transacted sessions or
temporary destinations. At such times the connection exception handler is called and the
application code has to catch the exception and restore state.

This section explains how automatic reconnection is enabled, how the broker behaves during a
reconnect, how automatic reconnection impacts producers and consumers, and how producers
and consumers should handle exceptions that result from connection failover. For additional information about this feature, please see “Connection Handling” in Oracle GlassFish Message Queue 4.4.2 Administration Guide.

Message Queue also provides a notification API that allows the client application to listen for closure and reconnection events and to respond to such events based on the notification type and the connection state. These notifications may be valuable in preparing the client for an impending event or for gathering diagnostic data. For more information, see “Connection Event Notification” on page 98.

Starting with version 4.1 of Message Queue, you can cluster brokers in either a conventional cluster or a high-availability cluster. The clustering model used may affect your client design. This section notes such design differences.

Enabling Auto-Reconnect

If you are using conventional clusters, you enable automatic reconnection by setting the connection factory `imqReconnectEnabled` attribute to `true`. If you are using a high availability cluster, the `imqReconnectEnabled` attribute is ignored; the client runtime will automatically reconnect to a backup broker if the connection is lost and not regained after no more than `imqReconnectAttempts` attempts. This applies to all deployment configurations: whether Message Queue is used stand alone or whether the connection is created through a resource adapter.

No matter which type of cluster you are using, you must also configure the connection factory administered object to specify the following information.

- **A list of message-service addresses** (using the `imqAddressList` attribute). Independently of the clustering model used, the client runtime uses this address list when it establishes the initial connection.

  When you connect to a conventional cluster, the client runtime also uses the address list when it tries to reestablish a connection to the message service: it attempts to connect to the brokers in the list until it finds (or fails to find) an available broker. If you specify only a single broker instance on the `imqAddressList` attribute, the configuration won’t support recovery from hardware failure.

  When you specify more than one broker, you can decide whether to use parallel brokers or a broker cluster. In a parallel configuration, there is no communication between brokers, while in a broker cluster, the brokers interact to distribute message delivery loads. (Refer to “Cluster Message Delivery” in Oracle GlassFish Message Queue 4.4.2 Technical Overview for more information on broker clusters.)

- **To enable parallel-broker reconnection**, set the `imqAddressListBehavior` attribute to `PRIORITY`. Typically, you would specify no more than a pair of brokers for this type of reconnection. This way, the messages are published to one broker, and all clients fail over together from the first broker to the second.
■ To enable clustered-broker reconnection, set the `imqAddressListBehavior` attribute to `RANDOM`. This way, the client runtime randomizes connection attempts across the list, and client connections are distributed evenly across the broker cluster.

Each broker in a cluster uses its own separate persistent store (which means that undelivered persistent messages are unavailable until a failed broker is back online). If one broker crashes, its client connections are reestablished on other brokers.

If you use the high availability clustering model, the address list is dynamically updated to include the brokers that are connected to the highly available database serving the cluster. In this case, the client runtime and the brokers use an internal protocol to determine which broker takes over the persistent data of the failed broker. Therefore the `imqAddressListBehavior` property does not apply to this model.

■ The number of iterations to be made over the list of brokers (using the `imqAddressListIterations` attribute) when attempting to create a connection or to reconnect.

For high-availability clusters, the broker will attempt to reconnect forever (no matter what value you specify for this attribute). If the client does not want this behavior, it must explicitly close the connection.

■ The number of attempts to reconnect to a broker if the first connection fails (using the `imqReconnectAttempts` attribute).

■ The interval, in milliseconds, between reconnect attempts, using the `imqReconnectInterval` attribute. This attribute applies to both clustering models.

### Single-Broker Auto-Reconnect

Configure your connection-factory object as follows:

**EXAMPLE 3-3 Example of Command to Configure a Single Broker**

```
imqobjmgr add -t cf -l "cn=myConnectionFactory" \  -o "imqAddressList=mq://jpgserv/jms" \  -o "imqReconnect=true" \  -o "imqReconnectAttempts=10" \  -j "java.naming.factory.initial = com.sun.jndi.fscontext.RefFSContextFactory" \  -j "java.naming.provider.url= file:///home/foo/imq_admin_objects"
```

This command creates a connection-factory object with a single address in the broker address list. If connection fails, the client runtime will try to reconnect with the broker 10 times. If an attempt to reconnect fails, the client runtime will sleep for three seconds (the default value for the `imqReconnectInterval` attribute) before trying again. After 10 unsuccessful attempts, the application will receive a `JMSException`.
You can ensure that the broker starts automatically at system start-up time. See “Starting Brokers Automatically” in Oracle GlassFish Message Queue 4.4.2 Administration Guide for information on how to configure automatic broker start-up. For example, on the Solaris platform, you can use /etc/rc.d scripts.

**Parallel Broker Auto-Reconnect**

Configure your connection-factory objects as follows:

**EXAMPLE 3–4**  Example of Command to Configure Parallel Brokers

```shell
imqobjmgr add -t cf -l "cn=myCF" \
   -o "imqAddressList=myhost1, mqtcp://myhost2:12345/jms" \
   -o "imqReconnect=true" \
   -o "imqReconnectRetries=5" \
   -j "java.naming.factory.initial = com.sun.jndi.fscontext.RefFSContextFactory" \
   -j "java.naming.provider.url= file:///home/foo/imq_admin_objects"
```

This command creates a connection factory object with two addresses in the broker list. The first address describes a broker instance running on the host myhost1 with a standard port number (7676). The second address describes a jms connection service running at a statically configured port number (12345).

**Clustered-Broker Auto-Reconnect**

Configure your connection-factory objects as follows:

**EXAMPLE 3–5**  Example of Command to Configure a Broker Cluster

```shell
imqobjmgr add -t cf -l "cn=myConnectionFactory" \
   -o "imqAddressList=mq://myhost1/ssljms, \
       mq://myhost2/ssljms, \
       mq://myhost3/ssljms, \
       mq://myhost4/ssljms" \
   -o "imqReconnect=true" \
   -o "imqReconnectRetries=5" \
   -o "imqAddressListBehavior=RANDOM" \
   -j "java.naming.factory.initial = com.sun.jndi.fscontext.RefFSContextFactory" \
   -j "java.naming.provider.url= file:///home/foo/imq_admin_objects"
```

This command creates a connection factory object with four addresses in the imqAddressList. All the addresses point to jms services running on SSL transport on different hosts. Since the imqAddressListBehavior attribute is set to RANDOM, the client connections that are established using this connection factory object will be distributed randomly among the four brokers in the
address list. If you are using a high availability cluster, the RANDOM attribute is ignored during a failover reconnect after losing an existing connection to a broker.

For a conventional cluster, you must configure one of the brokers in the cluster as the master broker. In the connection-factory address list, you can also specify a subset of all the brokers in the cluster.

**Auto-Reconnect Behaviors**

A broker treats an automatic reconnection as it would a new connection. When the original connection is lost, all resources associated with that connection are released. For example, in a broker cluster, as soon as one broker fails, the other brokers assume that the client connections associated with the failed broker are gone. After auto-reconnect takes place, the client connections are recreated from scratch.

Sometimes the client-side state cannot be fully restored by auto-reconnect. Perhaps a resource that the client needs cannot be recreated. In this case, the client runtime calls the client's connection exception handler and the client must take appropriate action to restore state. For additional information, see “Handling Exceptions When Failover Occurs” on page 106.

If the client is automatically-reconnected to a different broker instance, effects vary depending on the clustering model used.

- In a conventional cluster, persistent messages produced but not yet consumed may only be delivered to the consumer after the original broker recovers. Other state information held by the failed or disconnected broker can be lost. The messages held by the original broker, once it is restored, might be delivered out of order.
- In a high availability cluster, messages produced but not yet consumed continue to be delivered to the consumer without the original broker needing to recover.

A transacted session is the most reliable method of ensuring that a message isn’t lost if you are careful in coding the transaction. If auto-reconnect happens in the middle of a transaction, any attempt to produce or consume messages will cause the client runtime to throw a JMSException. In this case, applications must call `Session.rollback()` to roll back the transaction.

The Message Queue client runtime may throw a `TransactionRolledBackException` when `Session.commit()` is called during or after a failover occurs. In this case, the transaction is rolled back and a new transaction is automatically started. Applications are not required to call `Session.rollback()` to rollback the transaction after receiving a `TransactionRolledBackException`.

The Message Queue client runtime may throw a JMSException when `Session.commit()` is called during or after a failover occurs. In this case, the transaction state is unknown (may or may not be committed). Applications should call `Session.rollback()` to roll back the uncommitted transaction.
If you are using a high availability cluster, the only time your transaction might wind up in an unknown state is if it is not possible to reconnect to any brokers in the cluster. This should happen rarely if ever. For additional information, see "Handling Exceptions When Failover Occurs" on page 106.

Automatic reconnection affects producers and consumers differently:

- During reconnection, producers cannot send messages. The production of messages (or any operation that involves communication with the message broker) is blocked until the connection is reestablished.
- For consumers, automatic reconnection is supported for all client acknowledgment modes. After a connection is reestablished, the broker will redeliver all unacknowledged messages it had previously delivered, marking them with a `Redeliver` flag. The client can examine this flag to determine whether any message has already been consumed (but not yet acknowledged). In the case of nondurable subscribers, some messages might be lost because the broker does not hold their messages once their connections have been closed. Any messages produced for nondurable subscribers while the connection is down cannot be delivered when the connections is reestablished. For additional information, see "Handling Exceptions When Failover Occurs" on page 106.

### Auto-Reconnect Limitations

Notice the following points when using the auto-reconnect feature:

- Messages might be redelivered to a consumer after auto-reconnect takes place. In auto-acknowledge mode, you will get no more than one redelivered message. In other session types, all unacknowledged persistent messages are redelivered.
- While the client runtime is trying to reconnect, any messages sent by the broker to nondurable topic consumers are lost.
- Any messages that are in queue destinations and that are unacknowledged when a connection fails are redelivered after auto-reconnect. However, in the case of queues delivering to multiple consumers, these messages cannot be guaranteed to be redelivered to the original consumers. That is, as soon as a connection fails, an unacknowledged queue message might be rerouted to other connected consumers.
- In the case of a conventional broker cluster, the failure of the master broker prevents the following operations from succeeding on any other broker in the cluster:
  - Creating or destroying a new durable subscription.
  - Creating or destroying a new physical destination using the `imqcmd create dst` command.
  - Starting a new broker process. (However, the brokers that are already running continue to function normally even if the master broker goes down.)
You can configure the master broker to restart automatically using Message Queue broker support for rc scripts or the Windows service manager.

- Auto-reconnect doesn’t work if the client uses a ConnectionConsumer to consume messages. In that case, the client runtime throws an exception.

Handling Exceptions When Failover Occurs

Several kinds of exceptions can occur as a result of the client being reconnected after a failover. How the client application should handle these exceptions depends on whether a session is transacted, on the kind of exception thrown, and on the client’s role—as producer or consumer. The following sections discuss the implications of these factors.

Independently of how the exception is raised, the client application must never call System.exit() to exit the application because this would prevent the Message Queue client runtime from reconnecting to an alternate or restarted broker.

When a failover occurs, exception messages may be shown on the application’s console and recorded in the broker’s log. These messages are for information only. They may be useful in troubleshooting, but minimizing or eliminating the impact of a failover is best handled preemptively by the application client in the ways described in the following sections.

Note – Message Queue provides a notification API that allows the client application to listen for closure and reconnection events and to respond to such events based on the notification type and the connection state. These notifications may be valuable in preparing the client for an impending event or for gathering diagnostic data. For more information, see “Connection Event Notification” on page 98.

Handling Exceptions in a Transacted Session

A transacted session might fail to commit and (throw an exception) either because a failover occurs while statements within the transaction are being executed or because the failover occurs during the call to Session.commit(). In the first case, the failover is said to occur during an open transaction; in the second case, the failover occurs during the commit itself.

In the case of a failover during an open transaction, when the client application calls Session.commit(), the client runtime will throw a TransactionRolledBackException and roll back the transaction causing the following to happen.

- Messages that have been produced (but not committed) in the transacted session are discarded and not delivered to the consumer.
- All messages that have been consumed (but not committed) in the transacted session are redelivered to the consumer with the Redeliver flag set.
- A new transaction is automatically started.
If the client application itself had called Session.rollback after a failover (before the Session.commit is executed) the same things would happen as if the application had received a TransactionRollbackException. After receiving a TransactionRollbackException or calling Session.rollback(), the client application must retry the failed transaction. That is, it must re-send and re-consume the messages that were involved in the failed-over transaction.

In the second case, when the failover occurs during a call to Session.commit, there may be three outcomes:

- The transaction is committed successfully and the call to Session.commit does not return an exception. In this case, the application client does not have to do anything.
- The runtime throws a TransactionRolledbackException and does not commit the transaction. The transaction is automatically rolled back by the Message Queue runtime. In this case, the client application must retry the transaction as described for the case in which an open transaction is failed-over.
- A JMXException is thrown. This signals the fact that the transaction state is unknown: It might have either succeeded or failed. A client application should handle this case by assuming failure, pausing for three seconds, calling Session.rollback, and then retrying the operations. However, since the commit might have succeeded, when retrying the transacted operations, a producer should set application-specific properties on the messages it re-sends to signal that these might be duplicate messages. Likewise, consumers that retry receive operations should not assume that a message that is redelivered is necessarily a duplicate. In other words, to ensure once and only once delivery, both producers and consumers need to do a little extra work to handle this edge case. The code samples presented next illustrate good coding practices for handling this situation.

If you are using a high availability cluster, the only time this condition might arise is when the client is unable to connect to any backup broker. This should be extremely rare.

The next two examples illustrate how stand-alone Message Queue producers and consumers should handle transactions during a failover. To run the sample programs, do the following:

1. Start two high availability brokers. The brokers can be on the same machine or on different machines, but they must be in the same cluster.
2. Start the example programs. For example:
   ```bash
   java -DmqAddressList="localhost:777"
   test.jmsclient.ha.FailoverQSender
   ```
   ```bash
   java -DmqAddressList="localhost:777"
   test.jmsclient.ha.FailoverQReceiver
   ```

   It does not matter in what order you start the programs. The only property that you must specify is mqAddressList. The client application will be automatically failed over to a backup broker if the connection to its home broker fails. (The imqReconnectEnabled and imqAddressListIterations properties are ignored for a high availability cluster.)
3. Kill the broker to which the producing or consuming application is connected. The clients will reconnect, validate, and continue the failed transaction. A message produced or consumed in a transaction is either committed or rolled back after a successful failover.

4. You can restart the dead broker and retry the failover operation by killing the new home broker.

**Transacted Session: Failover Producer Example**

The following code sample shows the work that a producer in a transacted session needs to do to recover state after a failover. Note how the application tests both for rollback exceptions and for JMS exceptions. Note also the use of a counter to allow the producer and consumer to verify message order and delivery.

```java
package test.jmsclient.ha;
import java.util.Date;
import javax.jms.*;
import com.sun.messaging.jms.Connection;
import com.sun.messaging.jms.notification.*;

/**
 * This sample program uses a transacted session to send messages.
 * It is designed to run with test.jmsclient.ha.FailoverQReceiver
 * @version 1.0
 */
public class FailoverQSender
    implements ExceptionListener, EventListener, Runnable {
    //constant - commit property name
    public static final String COMMIT_PROPERTY_NAME = "COMMIT_PROPERTY";
    //constant - message counter
    public static final String MESSAGE_COUNTER = "counter";
    //constant - destination name
    public static final String TEST_DEST_NAME = "FailoverTestDest001";
    //queue connection
    QueueConnection conn = null;
    //session
    QueueSession session = null;
    //queue sender
```
QueueSender sender = null;
//queue destination
Queue queue = null;

//committed counter.
private int commitCounter = 0;
//current message counter
private int currentCounter = 0;
//set to true if the application is connected to the broker.
private boolean isConnected = false;

/**
 * Default constructor - do nothing.
 * Properties are passed in from init() method.
 */
public FailoverQSender() {

//set up JMS environment
setup();
}

/**
 * Connection Exception listener.
 */
public void onException (JMSException e) {

//The run() method will exit.
this.isConnected = false;

log ("Exception listener is called.
Connection is closed by MQ client runtime.");
log (e);
}

/**
 * this method is called when a MQ connection event occurred.
 */
public void onEvent (Event connectionEvent) {

log (connectionEvent);
}

/**
 * Rollback the application data.
 */
private void rollBackApplication() {

this.currentCounter = this.commitCounter;
}
```java
log ("Application rolled back., current (commit) counter: "+ currentCounter);
}
/**
 * Roll back the current jms session.
 */
private void rollBackJMS() {
    try {
        log("Rolling back JMS ...., commit counter: "+commitCounter);
        session.rollback();
    } catch (JMSException jmse) {
        log("Rollback failed");
        log(jmse);
        //application may decide to log and continue sending messages
        //without closing the application.
        close();
    }
}
/**
 * rollback application data and jms session.
 */
private void rollBackAll() {
    //rollback jms
    rollBackJMS();
    //rollback app data
    rollBackApplication();
}
/**
 * close JMS connection and stop the application
 */
private void close() {
    try {
        if ( conn != null ) {
            //close the connection
            conn.close();
        }
    } catch (Exception e) {
        //log exception
        log (e);
    } finally {
        //set flag to true. application thread will exit
    }
```
isConnected = false;
}
}
/**
 * Send messages in a loop until the connection is closed.
 * Session is committed for each message sent.
 */
public void run () {

//start producing messages
while (isConnected) {

    try {
    //reset message counter if it reaches max int value
    checkMessageCounter();
    //create a message
    Message m = session.createMessage();
    //get the current message counter value
    int messageCounter = this.getMessageCounter();
    //set message counter to message property
    m.setIntProperty(MESSAGE_COUNTER, messageCounter);
    //set commit property
    m.setBooleanProperty(COMMIT_PROPERTY_NAME, true);
    //send the message
    sender.send(m);

    log("Sending message: " + messageCounter + ", current connected broker: " + this.getCurrentConnectedBrokerAddress());

    //commit the message
    this.commit();

    // pause 3 seconds
    sleep(3000);
    } catch (TransactionRolledBackException trbe) {
    //rollback app data
    rollBackApplication();
    } catch (JMSException jmse) {
        if (isConnected == true) {
            //rollback app data and JMS session
            rollBackAll();
        }
    }
}
}
/**
 * Reset all counters if integer max value is reached.
 */
private void checkMessageCounter() {
    if (currentCounter == Integer.MAX_VALUE) {
        currentCounter = 0;
        commitCounter = 0;
    }
}

/**
 * Set up testing parameters - connection, destination, etc
 */
protected void setup() {
    try {
        // get connection factory
        com.sun.messaging.QueueConnectionFactory factory =
            new com.sun.messaging.QueueConnectionFactory();
        // create a queue connection
        conn = factory.createQueueConnection();
        // set exception listener
        conn.setExceptionListener(this);
        // set event listener
        ((com.sun.messaging.jms.Connection) conn).setEventListener(this);
        // get destination name
        String destName = TEST_DEST_NAME;
        // create a transacted session
        session = conn.createQueueSession(true, Session.AUTO_ACKNOWLEDGE);
        // get destination
        queue = session.createQueue(destName);
        // create queue sender
        sender = session.createSender(queue);
        // set isConnected flag to true.
        this.isConnected = true;
    } catch (JMSException jmse) {
        this.isConnected = false;
    }
}
/**
 * get the next message counter.
 */
private synchronized int getMessageCounter () {
    return ++ currentCounter;
}

/**
 * commit the current transaction/session.
 */
private void commit() throws JMSException {
    session.commit();
    this.commitCounter = currentCounter;
    log ("Transaction committed, commit counter: " + commitCounter);
}

/**
 * Get the current connected broker address.
 */
private StringgetCurrentConnectedBrokerAddress() {
    return ((com.sun.messaging.jms.Connection)conn).getBrokerAddress();
}

/**
 * log a string message.
 * @param msg
 */
private synchronized void log (String msg) {
    System.out.println(new Date() + ":");
    System.out.println(msg);
}

/**
 * Log an exception received.
 */
private synchronized void log (Exception e) {
    System.out.println(new Date() + ": Exception:");
    e.printStackTrace();
}

/**
 * Log the specified MQ event.
 */
private synchronized void log (Event event) {
    try {
        System.out.println(new Date() + ": Received MQ event notification.");
        System.out.println("*** Event code: " + event.getEventCode());
        System.out.println("*** Event message: " + event.getEventMessage());
    }
}
Transacted Session: Failover Consumer Example

The following code sample shows the work that a consumer in a transacted session needs to do in order to recover state after a failover. Note how the application tests both for rollback exceptions and JMS exceptions. Note also the use of a counter to allow the producer and consumer to verify message order and delivery.

```java
private void sleep (long millis) {
    try {
        Thread.sleep(millis);
    } catch (java.lang.InterruptedException inte) {
        log (inte);
    }
}

/**
 * The main program.
 */
public static void main (String args[]) {
    FailoverQSender fp = new FailoverQSender();
    fp.run();
}
```

Client Connection Failover (Auto-Reconnect)
public class FailoverQReceiver
    implements ExceptionListener, EventListener, Runnable {

    // queue connection
    private QueueConnection conn = null;
    // queue session
    private QueueSession session = null;
    // queue receiver
    private QueueReceiver qreceiver = null;
    // queue destination
    private Queue queue = null;

    // committed counter.
    private int commitCounter = 0;
    // flag to indicate if the connection is connected to the broker.
    private boolean isConnected = false;
    // flag to indicate if current connection is to HA broker cluster.
    private boolean isHAConnection = false;
    // application data holder.
    private Vector data = new Vector();

    /**
     * Default constructor - JMS setup.
     */
    public FailoverQReceiver() {
        // set up JMS environment
        setup();
    }

    /**
     * Connection Exception listener.
     */
    public void onException (JMSException e) {
        // The run() method will exit.
        this.isConnected = false;

        log ("Exception listener is called. Connection is closed
            by MQ client runtime.");
        log (e);
    }

    /**
     * log the connection event.
     */
    public void onEvent (Event connectionEvent) {

    }

    private void setup() {

    }

    private void log (Object s) {

    }

    private void log (Object s, Exception e) {

    }

    private void log (String s, Exception e) {

    }

    }
/**
 * Roll back application data.
 */
private void rollBackApplication() {
    // reset application data
    this.reset();

    log("Rolled back application data, current commit counter: "+commitCounter);
}

/**
 * Clear the application data for the current un-committed transaction.
 */
private void reset() {
    data.clear();
}

/**
 * Roll back JMS transaction and application.
 */
private void rollBackAll() {
    try {
        // rollback JMS
        rollBackJMS();
        // rollback application data
        rollBackApplication();
    } catch (Exception e) {
        log("rollback failed. closing JMS connection ... ");

        // application may decide NOT to close connection if rollback failed.
        close();
    }
}

/**
 * Roll back jms session.
 */
private void rollBackJMS() throws JMSException {
    session.rollback();
    log("JMS session rolled back ...., commit counter: "+commitCounter);
}
/**
   * Close JMS connection and exit the application.
   */
private void close() {
    try {
        if ( conn != null ) {
            conn.close();
        }
    } catch (Exception e) {
        log (e);
    } finally {
        isConnected = false;
    }
}

/**
   * Receive, validate, and commit messages.
   */
public void run () {
  //produce messages
  while (isConnected) {
    try {
      //receive message
      Message m = qreceiver.receive();
      //process message -- add message to the data holder
      processMessage(m);
      //check if the commit flag is set in the message property
      if ( shouldCommit(m) ) {
        //commit the transaction
        commit(m);
      }
    } catch (TransactionRolledBackException trbe) {
      log ("transaction rolled back by MQ ...");
      //rollback application data
      rollBackApplication();
    } catch (JMSException jmse) {
      //The exception can happen when receiving messages
      //and the connected broker is killed.
      if ( isConnected == true ) {
        //rollback MQ and application data
        rollBackAll();
      }
    } catch (Exception e) {

log (e);

// application may decide NOT to close the connection
// when an unexpected Exception occurred.
close();
}
}
}

/**
 * Set up testing parameters - connection, destination, etc
 */
protected void setup() {
    try {
        // get connection factory
        com.sun.messaging.QueueConnectionFactory factory =
            new com.sun.messaging.QueueConnectionFactory();

        // create jms connection
        conn = factory.createQueueConnection();

        // set exception listener
        conn.setExceptionListener(this);

        // set event listener
        ((com.sun.messaging.jms.Connection) conn).setEventListener(this);

        // test if this is a HA connection
        isHAConnection = ((com.sun.messaging.jms.Connection) conn).isConnectedToHABroker();
        log ("Is connected to HA broker cluster: " + isHAConnection);

        // get destination name
        String destName = FailoverQSender.TEST_DEST_NAME;

        // create a transacted session
        session = conn.createQueueSession(true, -1);

        // get destination
        queue = session.createQueue(destName);

        // create queue receiver
        qreceiver = session.createReceiver(queue);

        // set isConnected flag to true
        isConnected = true;

        // start the JMS connection
        conn.start();
        log("Ready to receive on destination: " + destName);
} catch (JMSException jmse) {
    isConnected = false;
    log (jmse);
    close();
}

/**
* Check if we should commit the transaction.
*/
private synchronized boolean shouldCommit(Message m) {

    boolean flag = false;

    try {
        //get the commit flag set by the FailoverQSender
        flag = m.getBooleanProperty(FailoverQSender.COMMIT_PROPERTY_NAME);

        if (flag) {
            //check if message property contains expected message counter
            validate (m);
        }
    } catch (JMSException jmse) {
        log (jmse);
    }

    return flag;
}

/**
* A very simple validation only. More logic may be added to validate
* message ordering and message content.
* @param m Message The last message received for the current transaction.
*/
private void validate (Message m) {

    try {
        //get message counter property
        int counter = m.getIntProperty(FailoverQSender.MESSAGE_COUNTER);
        //The counter is set sequentially and must be received in right order.
        //Each message is committed after validated.
        if (counter != (commitCounter + 1)) {
            this.printData();
            throw new RuntimeException("validation failed.");
        }

        log ("messages validated. ready to commit ...");
    } catch (JMSException jmse) {
        log (jmse);
    }

}}
} catch (JMSException jmse) {
    log (jmse);
    printData();
    throw new RuntimeException("Exception occurred during validation:
    " + jmse);
}

/**
 * Get the message counter and put it in the data holder.
 * @param m the current message received
 */
private synchronized void processMessage(Message m) throws JMSException {
    // get message counter. this value is set by the FailoverQSender.
    int ct = m.getIntProperty(FailoverQSender.MESSAGE_COUNTER);
    // log the message
    log("received message: " + ct
    
    
    +", current connected broker:
    
    " + this.getCurrentConnectedBrokerAddress());
    // saved the data in data holder.
    data.addElement(new Integer(ct));
}

/**
 * commit the current transaction.
 * @param m the last received message to be committed.
 * @throws JMSException if commit failed.
 */
private void commit(Message m) throws JMSException {
    //commit the transaction
    session.commit();
    //get the current message counter
    int counter = m.getIntProperty(FailoverQSender.MESSAGE_COUNTER);
    //set the commit counter
    commitCounter = counter;
    //clear app data
    this.reset();
    //log ("Messages committed, commitCounter: " + commitCounter);
    


/*
private synchronized void log (Exception e) {
    System.out.println(new Date() + ": Exception Stack Trace: ");
    e.printStackTrace();
}

private synchronized void log (Event event) {
    try {
        System.out.println(new Date() + ": Received MQ event notification.");
        System.out.println("*** Event Code: " + event.getEventCode());
        System.out.println("*** Event message: " + event.getEventMessage());
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private void log (String msg) {
    System.out.println(new Date() + ": "+ msg);
}

/*
 * Log the specified message.
 */
private void log (String msg) {
    System.out.println(new Date() + "":" + msg);
}

/*
 * print values stored in the data holder.
 */
private void printData() {

    for (int i=0; i< data.size(); i++) {
        log(" *** data index " + i + ": " + data.elementAt(i));
    }
}

private String getCurrentConnectedBrokerAddress() {
    return ((com.sun.messaging.jms.Connection)conn).getBrokerAddress();
}

/*
 * The main method. This starts the failover queue receiver.
 */
public static void main (String args[]) {
    FailoverQReceiver fqr = new FailoverQReceiver();
Handling Exceptions in a Non-Transacted Session

If a connection is failed-over for a producer in a non-transacted session, a client application may receive a JMSException. The application thread that receives the exception should pause for a few seconds and then resend the messages. The client application may want to set a flag on the resent messages to indicate that they could be duplicates.

If a connection is failed over for a message consumer, the consequences vary with the sessions acknowledge mode:

- In client-acknowledge mode, calling Message.acknowledge or MessageConsumer.receive during a failover will raise a JMSException. The consumer should call Session.recover to recover or re-deliver the unacknowledged messages and then call Message.acknowledge or MessageConsumer.receive.

- In auto-acknowledge mode, after getting a JMSException, the synchronous consumer should pause a few seconds and then call MessageConsumer.receive to continue receiving messages. Any message that failed to be acknowledged (due to the failover) will be redelivered with the redelivered flags set to true.

- In dups-OK-acknowledge mode, the synchronous consumer should pause a few seconds after getting an exception and then call MessageConsumer.receive to continue receiving messages. In this case, it's possible that messages delivered and acknowledged (before the failover) could be redelivered.

Failover Producer Example

The following code sample illustrates good coding practices for handling exceptions during a failover. It is designed to send non-transacted, persistent messages forever and to handle JMSExceptions when a failover occurs. The program is able to handle either a true or false setting for the imqReconnectEnabled property. To run the program enter one of the following commands.

```
java dura.example.FailoverProducer

java -DimqReconnectEnabled=true dura.example.FailoverProducer
```

/*
* @(#)FailoverProducer.java 1.1 06/06/09
* Copyright 2006 Sun Microsystems, Inc. All Rights Reserved
* SUN PROPRIETARY/CONFIDENTIAL
* Use is subject to license terms. */
package dura.example;
import javax.jms.*;
import com.sun.messaging.ConnectionConfiguration;
import java.util.*;

public class FailoverProducer implements ExceptionListener {

    //connection factory
    private com.sun.messaging.TopicConnectionFactory factory;
    //connection
    private TopicConnection pconn = null;
    //session
    private TopicSession psession = null;
    //publisher
    private TopicPublisher publisher = null;
    //topic
    private Topic topic = null;
    //This flag indicates whether this test client is closed.
    private boolean isClosed = false;
    //auto reconnection flag
    private boolean autoReconnect = false;
    //destination name for this example.
    private static final String DURA_TEST_TOPIC = "DuraTestTopic";
    //the message counter property name
    public static final String MESSAGE_COUNTER = "MESSAGE_COUNTER";
    //the message in-doubt-bit property name
    public static final String MESSAGE_IN_DOUBT = "MESSAGE_IN_DOUBT";

    /**
     * Constructor. Get imqReconnectEnabled property value from
     * System property.
     */
    public FailoverProducer () {
        try {
            autoReconnect =
                    Boolean.getBoolean(ConnectionConfiguration.imqReconnectEnabled);
        } catch (Exception e) {
            this.printStackTrace(e);
        }
    }

    /**
     * Connection is broken if this handler is called.
     * If autoReconnect flag is true, this is called only
     * if no more retries from MQ.
     */

public void onException (JMSException jmse) {
    this.printException (jmse);
}

/**
 * create MQ connection factory.
 * @throws JMSException
 */
private void initFactory() throws JMSException {
    //get connection factory
    factory = new com.sun.messaging.TopicConnectionFactory();
}

/**
 * JMS setup. Create a Connection,Session, and Producer.
 * 
 * If any of the JMS object creation fails (due to system failure),
 * it retries until it succeeds.
 * 
 */
private void initProducer() {
    boolean isConnected = false;
    while ( isClosed == false && isConnected == false ) {
        try {
            println("producer client creating connection ...");

            //create connection
            pconn = factory.createTopicConnection();

            //set connection exception listener
            pconn.setExceptionListener(this);

            //create topic session
            psession = pconn.createTopicSession(false,
                    Session.CLIENT_ACKNOWLEDGE);

            //get destination
            topic = psession.createTopic(DURA_TEST_TOPIC);

            //publisher
            publisher = psession.createPublisher(topic);

            //set flag to true
            isConnected = true;
        }
    }
}
println("producer ready.");
}
catch (Exception e) {

println("*** connect failed ... sleep for 5 secs.");

try {
   // close resources.
   if ( pconn != null ) {
      pconn.close();
   }
   // pause 5 secs.
   Thread.sleep(5000);

} catch (Exception e1) {
   
   }
}

/**
 * Start test. This sends JMS messages in a loop (forever).
 */
public void run () {

try {
   // create MQ connection factory.
   initFactory();
   
   // create JMS connection, session, and producer
   initProducer();
   
   // send messages forever.
   sendMessages();
} catch (Exception e) {
   this.printException(e);
}
}

/**
 * Send persistent messages to a topic forever. This shows how
 * to handle failover for a message producer.
 */
private void sendMessages() {

   // this is set to true if send failed.

boolean messageInDoubt = false;

// message to be sent
TextMessage m = null;

// msg counter
long msgcount = 0;

while (isClosed == false) {
    try {
        /**
         * create a text message
         */
        m = psession.createTextMessage();

        /**
         * the MESSAGE_IN_DOUBT bit is set to true if
         * you get an exception for the last message.
         */
        if (messageInDoubt == true) {
            m.setBooleanProperty(MESSAGE_IN_DOUBT, true);
            messageInDoubt = false;
            println("MESSAGE_IN_DOUBT bit is set to true
                     for msg: "+ msgcount);
        } else {
            m.setBooleanProperty(MESSAGE_IN_DOUBT, false);
        }

        // set message counter
        m.setLongProperty(MESSAGE_COUNTER, msgcount);

        // set message body
        m.setText("msg: "+ msgcount);

        // send the msg
        publisher.send(m, DeliveryMode.PERSISTENT, 4, 0);
        println("sent msg: "+ msgcount);

        /**
         * reset counter if reached max long value.
         */
        if (msgcount == Long.MAX_VALUE) {
            msgcount = 0;
        }
    }
}
println("Reset message counter to 0.");
}

//increase counter
msgcount ++;
Thread.sleep(1000);
}

} catch (Exception e) {
    if (isClosed == false) {
        //set in doubt bit to true.
        messageInDoubt = true;
        this.printException(e);
        //init producer only if auto reconnect is false.
        if (autoReconnect == false) {
            this.initProducer();
        }
    }
}

/**
 * Close this example program.
 */
public synchronized void close() {
    try {
        isClosed = true;
        pconn.close();
        notifyAll();
    } catch (Exception e) {
        this.printException(e);
    }
}

/**
 * print the specified exception.
 * @param e the exception to be printed.
 */
private void printException(Exception e) {
    System.out.println(new Date().toString());
e.printStackTrace();
private void println (String msg) {
    System.out.println(new Date() + " : " + msg);
}

public static void main (String args[]) {
    FailoverProducer fp = new FailoverProducer();
    fp.run();
}

Failover Consumer Example

The following code sample, FailoverConsumer, illustrates good coding practices for handling exceptions during a failover. The transacted session is able to receive messages forever. The program sets the auto reconnect property to true, requiring the Message Queue runtime to automatically perform a reconnect when the connected broker fails or is killed. It is designed to work with the dura.example.FailoverProducer, shown in the previous section.

To run this program enter the following command.

java dura.example.FailoverConsumer

import java.util.Date;
import javax.jms.Destination;
import javax.jms.ExceptionListener;
import javax.jms.JMSException;
import javax.jms.Connection;
import javax.jms.MessageConsumer;
import javax.jms.Session;
import javax.jms.Topic;
import javax.jms.TransactionRolledBackException;
import com.sun.messaging.ConnectionConfiguration;

public class FailoverConsumer implements ExceptionListener, Runnable {

    // JMS connection
    private Connection conn = null;
    // JMS session
    private Session session = null;
    // JMS Message consumer
    private MessageConsumer messageConsumer = null;
    // JMS destination.
    private Destination destination = null;
    // flag indicates whether this program should continue running.
    private boolean isConnected = false;
    // destination name.
    private static final String DURA_TEST_TOPIC = "DuraTestTopic";
    // the commit counter, for information only.
    private long commitCounter = 0;

    /**
     * message counter property set by the producer.
     */
    public static final String MESSAGE_COUNTER = "MESSAGE_COUNTER";

    /**
     * Message in doubt bit set by the producer
     */
    public static final String MESSAGE_IN_DOUBT = "MESSAGE_IN_DOUBT";

    /**
     * receive time out
     */
    public static final long RECEIVE_TIMEOUT = 0;

    /**
     * Default constructor -
     * Set up JMS Environment.
     */
    public FailoverConsumer() {
        setup();
    }

    /* Connection Exception listener. This is called when connection
     * breaks and no reconnect attempts are performed by MQ client runtime.
     */
public void onException (JMSException e) {
    print ("Reconnect failed. Shutting down the connection ...");
    
    /**
     * Set this flag to false so that the run() method will exit.
     */
    this.isConnected = false;
    e.printStackTrace();
}

/**
 * Best effort to roll back a jms session. When a broker crashes, an
 * open-transaction should be rolled back. But the re-started broker
 * may not have the uncommitted transection information due to system
 * failure. In a situation like this, an application can just quit
 * calling rollback after retrying a few times. The uncommitted
 * transaction (resources) will eventually be removed by the broker.
 */
private void rollBackJMS() {

    //rollback fail count
    int failCount = 0;

    boolean keepTrying = true;

    while ( keepTrying ) {
        try {
            print ("<<< rolling back JMS ..... consumer commit counter: " + this.commitCounter);

            session.rollback();

            print("<<< JMS rolled back ..... consumer commit counter: " + this.commitCounter);
            keepTrying = false;
        }
        catch (JMSException jmse) {

            failCount ++;
            jmse.printStackTrace();

            sleep (3000); //3 secs

            if ( failCount == 1 ) {  

                print ("<<< rollback failed : total count" + failCount);
            }
        }
    }
}
keepTrying = false;
}
}
{
/**
 * Close the JMS connection and exit the program.
 */
private void close() {
  try {
    if ( conn != null ) {
      conn.close();
    }
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    this.isConnected = false;
  }
}
/*Receive messages in a loop until closed.*/
public void run () {
  while (isConnected) {
    try {
      /*receive message with specified timeout.*/
      Message m = messageConsumer.receive(RECEIVE_TIMEOUT);
      /* process the message. */
      processMessage(m);
      /* commit JMS transaction. */
      this.commit();
      /*increase the commit counter.*/
      this.commitCounter ++;
    } catch (TransactionRolledBackException trbe) {

  }
}
/**
 * the transaction is rolled back
 * a new transaction is automatically started.
 */
trbe.printStackTrace();
}
catch (JMSException jmse) {

    /* The transaction is in unknown state.
    * We need to roll back the transaction.*/
    jmse.printStackTrace();

    /* roll back if not closed.
    */
    if ( this.isConnected == true ) {
        this.rollBackJMS();
    }
}
catch (Exception e) {
    e.printStackTrace();

    /* Exit if this is an unexpected Exception.*/
    this.close();
}
} finally {
    ;//do nothing
}
}

print(" <<< consumer exit ...");
}

/*Set up connection, destination, etc*/
/
protected void setup() {
try {

    //create connection factory
    com.sun.messaging.ConnectionFactory factory =
    new com.sun.messaging.ConnectionFactory();

    //set auto reconnect to true.
    factory.setProperty(ConnectionConfiguration.imqReconnectEnabled, "true");
    //A value of -1 will retry forever if connection is broken.
    factory.setProperty(ConnectionConfiguration.imqReconnectAttempts, ",-1");
/**
 * Process the received message message.
 * This prints received message counter.
 * @param m the message to be processed.
 */
private synchronized void processMessage(Message m) {
  try {
    //in this example, we do not expect a timeout, etc.
    if (m == null) {
      throw new RuntimeException("<<< Received null message. Maybe reached max time out.");
    }
  } catch (JMSException jmse) {
    this.isConnected = false;
    jmse.printStackTrace();
    this.close();
  }
}

//retry interval - every 10 seconds
factory.setProperty(ConnectionConfiguration.imqReconnectInterval, "10000");
//create connection
conn = factory.createConnection();
//set client ID
conn.setClientID(DURA_TEST_TOPIC);

//set exception listener
conn.setExceptionListener(this);

//create a transacted session
session = conn.createSession(true, -1);

//get destination
destination = session.createTopic(DURA_TEST_TOPIC);

//message consumer
messageConsumer = session.createDurableSubscriber((Topic)destination, DURA_TEST_TOPIC);

//set flag to true
this.isConnected = true;
//we are ready, start the connection
conn.start();

print("<<< Ready to receive on destination: "+ DURA_TEST_TOPIC);

} catch (JMSException jmse) {
  this.isConnected = false;
  jmse.printStackTrace();
  this.close();
}

/**
 * Process the received message message.
 * This prints received message counter.
 * @param m the message to be processed.
 */
private synchronized void processMessage(Message m) {
  try {
    //in this example, we do not expect a timeout, etc.
    if (m == null) {
      throw new RuntimeException("<<< Received null message. Maybe reached max time out.");
    }
  } catch (JMSException jmse) {
    this.isConnected = false;
    jmse.printStackTrace();
    this.close();
  }
}
//get message counter property
long msgCtr = m.getLongProperty (MESSAGE_COUNTER);

//get message in-doubt bit
boolean indoubt = m.getBooleanProperty(MESSAGE_IN_DOUBT);

if ( indoubt) {
    print("<<< received message: " + msgCtr + ", indoubt bit is true");
} else {
    print("<<< received message: " + msgCtr);
}

} catch (JMSException jmse) {
    jmse.printStackTrace();
}

/**
 * Commit a JMS transaction.
 * @throws JMSException
 */
private void commit() throws JMSException {
    session.commit();
}

/**
 * Sleep for the specified time.
 * @param millis sleep time in milli-seconds.
 */
private void sleep (long millis) {
    try {
        Thread.sleep(millis);
    } catch (java.lang.InterruptedException inte) {
        print (inte);
    }
}

/**
 * Print the specified message.
 * @param msg the message to be printed.
 */
private static void print (String msg) {
    System.out.println(new Date() +": " + msg);
}

/**
 * Print Exception stack trace.
 * @param e the exception to be printed.
 */
Custom Client Acknowledgment

Message Queue supports the standard JMS acknowledgment modes (auto-acknowledge, client-acknowledge, and dups-OK-acknowledge). When you create a session for a consumer, you can specify one of these modes. Your choice will affect whether acknowledgment is done explicitly (by the client application) or implicitly (by the session) and will also affect performance and reliability. This section describes additional options you can use to customize acknowledgment behavior:

- You can customize the JMS client-acknowledge mode to acknowledge one message at a time.
- If performance is key and reliability is not a concern, you can use the proprietary no-acknowledge mode to have the broker consider a message acknowledged as soon as it has been sent to the consuming client.

The following sections explain how you program these options.

Using Client Acknowledge Mode

For more flexibility, Message Queue lets you customize the JMS client-acknowledge mode. In client-acknowledge mode, the client explicitly acknowledges message consumption by invoking the `acknowledge()` method of a message object. The standard behavior of this method is to cause the session to acknowledge all messages that have been consumed by any consumer in the session since the last time the method was invoked. (That is, the session acknowledges the current message and all previously unacknowledged messages, regardless of who consumed them.)

In addition to the standard behavior specified by JMS, Message Queue lets you use client-acknowledge mode to acknowledge one message at a time.
Observe the following rules when implementing custom client acknowledgment:

- To acknowledge an individual message, call the `acknowledgeThisMessage()` method. To acknowledge all messages consumed so far, call the `acknowledgeUpThroughThisMessage()` method. Both are shown in the following code example.

  ```java
  public interface com.sun.messaging.jms.Message {
      void acknowledgeThisMessage() throws JMSException;
      void acknowledgeUpThroughThisMessage() throws JMSException;
  }
  ```

- When you compile the resulting code, include both `imq.jar` and `jms.jar` in the class path.

- Don’t call `acknowledge()`, `acknowledgeThisMessage()`, or `acknowledgeUpThroughThisMessage()` in any session except one that uses client-acknowledge mode. Otherwise, the method call is ignored.

- Don’t use custom acknowledgment in transacted sessions. A transacted session defines a specific way to have messages acknowledged.

If a broker fails, any message that was not acknowledged successfully (that is, any message whose acknowledgment ended in a `JMSException`) is held by the broker for delivery to subsequent clients.

**Example 3–6** demonstrates both types of custom client acknowledgment.

**EXAMPLE 3–6** Example of Custom Client Acknowledgment Code

```java
... import javax.jms.*;
...// [Look up a connection factory and create a connection.]
    Session session = connection.createSession(false,
        Session.CLIENT_ACKNOWLEDGE);

...// [Create a consumer and receive messages.]
    Message message1 = consumer.receive();
    Message message2 = consumer.receive();
    Message message3 = consumer.receive();

...// [Process messages.]

...// [Acknowledge one individual message.
    Notice that the following acknowledges only message 2.]
    ((com.sun.messaging.jms.Message)message2).acknowledgeThisMessage();

...// [Continue. Receive and process more messages.]
```
Using No Acknowledge Mode

No-acknowledge mode is a nonstandard extension to the JMS API. Normally, the broker waits for a client acknowledgment before considering that a message has been acknowledged and discarding it. That acknowledgment must be made programatically if the client has specified client-acknowledge mode or it can be made automatically, by the session, if the client has specified auto-acknowledge or dups-OK-acknowledge. If a consuming client specifies no-acknowledge mode, the broker discards the message as soon as it has sent it to the consuming client. This feature is intended for use by nondurable subscribers consuming nonpersistent messages, but it can be used by any consumer.

Using this feature improves performance by reducing protocol traffic and broker work involved in acknowledging a message. This feature can also improve performance for brokers dealing with misbehaving clients who do not acknowledge messages and therefore tie down broker memory resources unnecessarily. Using this mode has no effect on producers.

You use this feature by specifying NO_ACKNOWLEDGE for the acknowledgeMode parameter to the createSession, createQueueSession, or createTopicSession method. No-acknowledge mode must be used only with the connection methods defined in the com.sun.messaging.jms package. Note however that the connection itself must be created using the javax.jms package.

The following are sample variable declarations for connection, queueConnection and topicConnection:

```java
javax.jms.Connection connection;
javax.jms.QueueConnection queueConnection
javax.jms.TopicConnection topicConnection
```

The following are sample statements to create different kinds of no-acknowledge sessions:
//to create a no ack session
Session noAckSession =
    ((com.sun.messaging.jms.Connection)connection)
    .createSession(com.sun.messaging.jms.Session.NO_ACKNOWLEDGE);

// to create a no ack topic session
TopicSession noAckTopicSession =
    ((com.sun.messaging.jms.TopicConnection) topicConnection)
    .createTopicSession(com.sun.messaging.jms.Session.NO_ACKNOWLEDGE);

//to create a no ack queue session
QueueSession noAckQueueSession =
    ((com.sun.messaging.jms.QueueConnection) queueConnection)
    .createQueueSession(com.sun.messaging.jms.Session.NO_ACKNOWLEDGE);

Specifying no-acknowledgment mode for a session results in the following behavior:

- The client runtime will throw a JMSException if Session.recover() is called.
- The client runtime will ignore a call to the Message.acknowledge() method from a consumer.
- Messages can be lost. As opposed to dups-OK-acknowledge, which can result in duplicate messages being sent, no-acknowledge mode bypasses checks and balances built into the system and may result in message loss.

Schema Validation of XML Payload Messages

This Message Queue feature enables validation of the content of a text (not object) XML message against an XML schema at the point the message is sent to the broker.

When XML validation is enabled, the Message Queue client runtime will attempt to validate an XML message against specified XSDs before sending the message to the broker. The location of the XML schema (XSD) is specified as a property of a Message Queue destination. If the specified schema cannot be located or the message cannot be validated, the message is not sent, and an exception is thrown.

If no XSD location is specified, the DTD declaration within the XML document is used to perform DTD validation. (XSD validation, which includes data type and value range validation, is more rigorous than DTD validation.)

Client applications using this feature should upgrade Java SE version to JRE 1.5 or above.

XML schema validation is enabled using the following physical destination properties: validateXMLSchemaEnabled, XMLSchemaURIList, and reloadXMLSchemaOnFailure. These properties are described in Chapter 18, “Physical Destination Property Reference,” in Oracle GlassFish Message Queue 4.4.2 Administration Guide. The property values can be set at
destination creation or update time by using the `imqcmd create dst` or `imqcmd update dst` command, respectively. The XML validation properties should be set when a destination is inactive: that is, when it has no consumers and producers, and when there are no messages in the destination.

If any of the XML validation properties are set while a destination is active (for example, if a producer is connected to the destination), the change will not take effect until the producer reconnects to the broker. Similarly, if an XSD is changed, as a result of changing application requirements, all client applications producing XML messages based on the changed XSD must reconnect to the broker.

If the `reloadXMLSchemaOnFailure` property is set to `true` and XML validation fails, then the Message Queue client runtime will attempt to reload the XSD before attempting again to validate a message. The client runtime will throw an exception if the validation fails using the reloaded XSD.

**Communicating with C Clients**

Message Queue supports C clients as message producers and consumers.

A Java client consuming messages sent by a C client faces only one restriction: a C client cannot be part of a distributed transaction, and therefore a Java client receiving a message from a C client cannot participate in a distributed transaction either.

A Java client producing messages for a consuming C client must be aware of the following differences in the Java and C interfaces because these differences will affect the C client’s ability to consume messages: C clients

- Can only consume messages of type `text` and `bytes`
- Cannot consume messages whose body has been compressed
- Cannot participate in distributed transactions
- Cannot receive SOAP messages

**Client Runtime Logging**

This section describes Message Queue 4.0 support for client runtime logging of connection and session-related events.

JDK 1.4 (and above) includes the `java.util.logging` library. This library implements a standard logger interface that can be used for application-specific logging.

The Message Queue client runtime uses the Java Logging API to implement its logging functions. You can use all the J2SE 1.4 logging facilities to configure logging activities. For example, an application can use the following Java logging facilities to configure how the Message Queue client runtime outputs its logging information:
Logging Name Spaces, Levels, and Activities

The Message Queue provider defines a set of logging name spaces associated with logging levels and logging activities that allow Message Queue clients to log connection and session events when a logging configuration is appropriately set.

The root logging name space for the Message Queue client runtime is defined as javax.jms. All loggers in the Message Queue client runtime use this name as the parent name space.

The logging levels used for the Message Queue client runtime are the same as those defined in the java.util.logging.Level class. This class defines seven standard log levels and two additional settings that you can use to turn logging on and off.

- OFF              Turns off logging.
- SEVERE           Highest priority, highest value. Application-defined.
- WARNING          Application-defined.
- INFO             Application-defined.
- CONFIG           Application-defined
- FINE             Application-defined.
- FINER            Application-defined.
- FINEST           Lowest priority, lowest value. Application-defined.
- ALL              Enables logging of all messages.

In general, exceptions and errors that occur in the Message Queue client runtime are logged by the logger with the javax.jms name space.

- Exceptions thrown from the JVM and caught by the client runtime, such as IOException, are logged by the logger with the logging name space javax.jms at level WARNING.
- JMS exceptions thrown from the client runtime, such as IllegalStateException, are logged by the logger with the logging name space javax.jms at level FINER.
- Errors thrown from the JVM and caught by the client runtime, such as OutOfMemoryError, are logged by the logger with the logging name space javax.jms at level SEVERE.
The following tables list the events that can be logged and the log level that must be set to log events for JMS connections and for sessions.

The following table describes log levels and events for connections.

<table>
<thead>
<tr>
<th>Log Level</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINE</td>
<td>Connection created</td>
</tr>
<tr>
<td>FINE</td>
<td>Connection started</td>
</tr>
<tr>
<td>FINE</td>
<td>Connection closed</td>
</tr>
<tr>
<td>FINE</td>
<td>Connection broken</td>
</tr>
<tr>
<td>FINE</td>
<td>Connection reconnected</td>
</tr>
<tr>
<td>FINER</td>
<td>Miscellaneous connection activities such as setClientID</td>
</tr>
<tr>
<td>FINEST</td>
<td>Messages, acknowledgments, Message Queue action and control messages (like committing a transaction)</td>
</tr>
</tbody>
</table>

For sessions, the following information is recorded in the log record.

- Each log record for a message delivered to a consumer includes ConnectionID, SessionID, and ConsumerID.
- Each log record for a message sent by a producer includes ConnectionID, SessionID, ProducerID, and destination name.

The table below describes log levels and events for sessions.

<table>
<thead>
<tr>
<th>Log Level</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>FINE</td>
<td>Session created</td>
</tr>
<tr>
<td>FINE</td>
<td>Session closed</td>
</tr>
<tr>
<td>FINE</td>
<td>Producer created</td>
</tr>
<tr>
<td>FINE</td>
<td>Consumer created</td>
</tr>
<tr>
<td>FINE</td>
<td>Destination created</td>
</tr>
<tr>
<td>FINER</td>
<td>Miscellaneous session activities such as committing a session.</td>
</tr>
<tr>
<td>FINEST</td>
<td>Messages produced and consumed. (Message properties and bodies are not logged in the log records.)</td>
</tr>
</tbody>
</table>
By default, the output log level is inherited from the JRE in which the application is running. Check the JRE_DIRECTORY/lib/logging.properties file to determine what that level is.

You can configure logging programatically or by using configuration files, and you can control the scope within which logging takes place. The following sections describe these possibilities.

**Using the JRE Logging Configuration File**

The following example shows how you set logging name spaces and levels in the JRE_DIRECTORY/lib/logging.properties file, which is used to set the log level for the Java runtime environment. All applications using this JRE will have the same logging configuration. The sample configuration below sets the logging level to INFO for the javax.jms.connection name space and specifies that output be written to java.util.logging.ConsoleHandler.

```properties
#logging.properties file.
# "handlers" specifies a comma separated list of log Handler
# classes. These handlers will be installed during VM startup.
# Note that these classes must be on the system classpath.
# By default we only configure a ConsoleHandler, which will only
# show messages at the INFO and above levels.

handlers= java.util.logging.ConsoleHandler

# Default global logging level.
# This specifies which kinds of events are logged across
# all loggers. For any given facility this global level
# can be overridden by a facility-specific level.
# Note that the ConsoleHandler also has a separate level
# setting to limit messages printed to the console.

.level= INFO

# Limit the messages that are printed on the console to INFO and above.

java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

# The logger with javax.jms.connection name space will write
# Level.INFO messages to its output handler(s). In this configuration
# the output handler is set to java.util.logging.ConsoleHandler.

javax.jms.connection.level = INFO
```

Client Runtime Logging
Using a Logging Configuration File for a Specific Application

You can also define a logging configuration file from the java command line that you use to run an application. The application will use the configuration defined in the specified logging file. In the following example, configFile uses the same format as defined in the JRE_DIRECTORY/lib/logging.properties file.

```
java -Djava.util.logging.config.file=configFile MQApplication
```

Setting the Logging Configuration Programmatically

The following code uses the java.util.logging API to log connection events by changing the javax.jms.connection name space log level to FINE. You can include such code in your application to set logging configuration programatically.

```
import java.util.logging.*;
//construct a file handler and output to the mq.log file
//in the system's temp directory.

    Handler fh = new FileHandler("%t/mq.log");
    fh.setLevel (Level.FINE);

    //Get Logger for "javax.jms.connection" domain.

    Logger logger = Logger.getLogger("javax.jms.connection");
    logger.addHandler (fh);
    logger.setLevel (Level.FINE);

//javax.jms.connection logger would log activities
//with level FINE and above.

    logger.setLevel (Level.FINE);
```
Using the Metrics Monitoring API

Message Queue provides several ways of obtaining metrics data as a means of monitoring and tuning performance. One of these methods, message-based monitoring, allows metrics data to be accessed programmatically and then to be processed in whatever way suits the consuming client. Using this method, a client subscribes to one or more metrics destinations and then consumes and processes messages produced by the broker to those destinations. Message-based monitoring is the most customized solution to metrics gathering, but it does require the effort of writing a consuming client that retrieves and processes metrics messages.

The methods for obtaining metrics data are described in Chapter 13, “Monitoring Broker Operations,” in Oracle GlassFish Message Queue 4.4.2 Administration Guide, which also discusses the relative merits of each method and the set of data that is captured by each. Before you decide to use message-based monitoring, you should consult this guide to make sure that you will be able to obtain the information you need using this method.

Message-based monitoring is enabled by the combined efforts of administrators and programmers. The administrator is responsible for configuring the broker so that it produces the messages of interest at a specified interval and that it persists these messages for a set time. The programmer is responsible for selecting the data to be produced and for creating the client that will consume and process the data.

This chapter focuses on the work the programmer must do to implement a message-based monitoring client. It includes the following sections:

- “Monitoring Overview” on page 146
- “Creating a Metrics-Monitoring Client” on page 148
- “Format of Metrics Messages” on page 149
- “Metrics Monitoring Client Code Examples” on page 153
Monitoring Overview

Message Queue includes an internal client that is enabled by default to produce different types of metrics messages. Production is actually enabled when a client subscribes to a topic destination whose name matches one of the metrics message types. For example, if a client subscribes to the topic `mq.metrics.jvm`, the client receives information about JVM memory usage.

The metrics topic destinations (metric message types) are described in Table 4–1.

<table>
<thead>
<tr>
<th>Topic Destination Name</th>
<th>Type of Metrics Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>mq.metrics.broker</td>
<td>Broker metrics: information on connections, message flow, and volume of messages in the broker.</td>
</tr>
<tr>
<td>mq.metrics.jvm</td>
<td>Java Virtual Machine metrics: information on memory usage in the JVM.</td>
</tr>
<tr>
<td>mq.metrics.destination_list</td>
<td>A list of all destinations on the broker, and their types.</td>
</tr>
<tr>
<td>mq.metrics.destination.queue.dn</td>
<td>Destination metrics for a queue of the specified name. Metrics data includes number of consumers, message flow or volume, disk usage, and more. Specify the destination name for the <code>dn</code> variable.</td>
</tr>
<tr>
<td>mq.metrics.destination.topic.dn</td>
<td>Destination metrics for a topic of the specified name. Metrics data includes number of consumers, message flow or volume, disk usage, and more. Specify the destination name for the <code>dn</code> variable.</td>
</tr>
</tbody>
</table>

A metrics message that is produced to one of the destinations listed in Table 4–1 is a normal JMS message; its header and body are defined to hold the following information:

- The message header has several properties, one that specifies the metrics message type, one that records the time the message was produced (timestamp), and a collection of properties identifying the broker that sent the metric message (broker host, port, and address/URL).
- The message body contains name-value pairs that vary with the message type.

  The section “Format of Metrics Messages” on page 149 provides complete information about the types of metrics messages and their content (name-value pairs).

To receive metrics messages, the consuming client must be subscribed to the destination of interest. Otherwise, consuming a metrics message is exactly the same as consuming any JMS message. The message can be consumed synchronously or asynchronously, and then processed as needed by the client.
Message-based monitoring is concerned solely with obtaining metrics information. It does not include methods that you can call to work with physical destinations, configure or update the broker, or shutdown and restart the broker.

**Administrative Tasks**

By default the Message Queue metrics-message producing client is enabled to produce nonpersistent messages every sixty seconds. The messages are allowed to remain in their respective destinations for 5 minutes before being automatically deleted. To persist metrics messages, to change the interval at which they are produced, or to change their time-to-live interval, the administrator must set the following properties in the `config.properties` file:

- `imq.metrics.topic.persist`
- `imq.metrics.topic.interval`
- `imq.metrics.topic.timetolive`

In addition, the administrator might want to set access controls on the metrics destinations. This restricts access to sensitive metrics data and helps limit the impact of metrics subscriptions on overall performance. For more information about administrative tasks in enabling message-based monitoring and access control, see “Using the Message-Based Monitoring API” in *Oracle GlassFish Message Queue 4.4.2 Administration Guide*.

**Implementation Summary**

The following task list summarizes the steps required to implement message based monitoring:

1. **To Implement Message-Based Monitoring**
   - The developer designs and writes a client that subscribes to one or more metrics destinations.
   - The administrator sets those metrics-related broker properties whose default values are not satisfactory.
   - *(Optional)* The administrator sets entries in the `access.control.properties` file to restrict access to metrics information.
   - The developer or the administrator starts the metrics monitoring client.

   When consumers subscribe to a metrics topic, the topic’s physical destination is automatically created. After the metrics topic has been created, the broker’s metrics message producer begins to send metrics messages to the appropriate destination.
Creating a Metrics-Monitoring Client

You create a metrics monitoring client in the same way that you would write any JMS client, except that the client must subscribe to one or more special metrics message topic and must be ready to receive and process messages of a specific type and format.

No hierarchical naming scheme is implied in the metrics-message names. You can’t use a wildcard character (*) to identify multiple destination names.

A client that monitors broker metrics must perform the following basic tasks:

▼ To Monitor Broker Metrics

1. Create a TopicConnectionFactory object.
2. Create a TopicConnection to the Message Queue service.
3. Create a TopicSession.
4. Create a metrics Topic destination object.
5. Create a TopicSubscriber.
6. Register as an asynchronous listener to the topic, or invoke the synchronous receive() method to wait for incoming metrics messages.
7. Process metrics messages that are received.

In general, you would use JNDI lookups of administered objects to make your client code provider-independent. However, the metrics-message production is specific to Message Queue, there is no compelling reason to use JNDI lookups. You can simply instantiate these administered objects directly in your client code. This is especially true for a metrics destination for which an administrator would not normally create an administered object.

Notice that the code examples in this chapter instantiate all the relevant administered objects directly.

You can use the following code to extract the type (String) or timestamp (long) properties in the message header from the message:

```java
MapMessage mapMsg;
/*
 * mapMsg is the metrics message received
 */
String type = mapMsg.getStringProperty("type");
long timestamp = mapMsg.getLongProperty("timestamp");
```
You use the appropriate get method in the class `javax.jms.MapMessage` to extract the name-value pairs. The get method you use depends on the value type. Three examples follow:

```java
long value1 = mapMsg.getLong("numMsgsIn");
long value2 = mapMsg.getLong("numMsgsOut");
int value3 = mapMsg.getInt("diskUtilizationRatio");
```

## Format of Metrics Messages

In order to consume and process a metrics message, you must know its type and format. This section describes the general format of metrics messages and provides detailed information on the format of each type of metrics message.

Metrics messages are of type `MapMessage`. (A type of message whose body contains a set of name-value pairs. The order of entries is not defined.)

- The message header has properties that are useful to applications. The `type` property identifies the type of metric message (and therefore its contents). It is useful if the same subscriber processes more than one type of metrics message for example, messages from the topics `mq.metrics.broker` and `mq.metrics.jvm`. The `timestamp` property indicates when the metric sample was taken and is useful for calculating rates or drawing graphs. The `brokerHost`, `brokerPort`, and `brokerAddress` properties identify the broker that sent the metric message and are useful when a single application needs to process metric messages from different brokers.

- The body of the message contains name-value pairs, and the data values depend on the type of metrics message. The following subsections describe the format of each metrics message type.

Note that the names of name-value pairs (used in code to extract data) are case-sensitive and must be coded exactly as shown. For example, `NumMsgsOut` is incorrect; `numMsgsOut` is correct.

### Broker Metrics

The messages you receive when you subscribe to the topic `mq.metrics.broker` have the type property set to `mq.metrics.broker` in the message header and have the data listed in Table 4-2 in the message body.

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>Value Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numConnections</td>
<td>long</td>
<td>Current number of connections to the broker</td>
</tr>
<tr>
<td>numMsgsIn</td>
<td>long</td>
<td>Number of JMS messages that have flowed into the broker since it was last started</td>
</tr>
</tbody>
</table>
TABLE 4-2  Data in the Body of a Broker Metrics Message  (Continued)

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>ValueType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numMsgsOut</td>
<td>long</td>
<td>Number of JMS messages that have flowed out of the broker since it was last started</td>
</tr>
<tr>
<td>numMsgs</td>
<td>long</td>
<td>Current number of JMS messages stored in broker memory and persistent store</td>
</tr>
<tr>
<td>msgBytesIn</td>
<td>long</td>
<td>Number of JMS message bytes that have flowed into the broker since it was last started</td>
</tr>
<tr>
<td>msgBytesOut</td>
<td>long</td>
<td>Number of JMS message bytes that have flowed out of the broker since it was last started</td>
</tr>
<tr>
<td>totalMsgBytes</td>
<td>long</td>
<td>Current number of JMS message bytes stored in broker memory and persistent store</td>
</tr>
<tr>
<td>numPktssIn</td>
<td>long</td>
<td>Number of packets that have flowed into the broker since it was last started; this includes both JMS messages and control messages</td>
</tr>
<tr>
<td>numPktssOut</td>
<td>long</td>
<td>Number of packets that have flowed out of the broker since it was last started; this includes both JMS messages and control messages</td>
</tr>
<tr>
<td>pktBytesIn</td>
<td>long</td>
<td>Number of packet bytes that have flowed into the broker since it was last started; this includes both JMS messages and control messages</td>
</tr>
<tr>
<td>pktBytesOut</td>
<td>long</td>
<td>Number of packet bytes that have flowed out of the broker since it was last started; this includes both JMS messages and control messages</td>
</tr>
<tr>
<td>numDestinations</td>
<td>long</td>
<td>Current number of destinations in the broker</td>
</tr>
</tbody>
</table>

JVM Metrics

The messages you receive when you subscribe to the topic `mq.metrics.jvm` have the type property set to `mq.metrics.jvm` in the message header and have the data listed in Table 4-3 in the message body.

TABLE 4-3  Data in the Body of a JVM Metrics Message

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>ValueType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>freeMemory</td>
<td>long</td>
<td>Amount of free memory available for use in the JVM heap</td>
</tr>
<tr>
<td>maxMemory</td>
<td>long</td>
<td>Maximum size to which the JVM heap can grow</td>
</tr>
<tr>
<td>totalMemory</td>
<td>long</td>
<td>Total memory in the JVM heap</td>
</tr>
</tbody>
</table>
Destination-List Metrics

The messages you receive when you subscribe to a topic named `mq.metrics.destination_list` have the type property set to `mq.metrics.destination_list` in the message header.

The message body contains a list of map names. Each destination on the broker is specified by a unique map name (a name-value pair) in the message body. The type of the name-value pair is `hashtable`.

The `name` (in the name-value pair) depends on whether the destination is a queue or a topic, and is constructed as follows:

- `mq.metrics.destination.queue.monitored_destination_name`
- `mq.metrics.destination.topic.monitored_destination_name`

The `value` (in the name-value pair) is an object of type `java.util.Hashtable`. This hashtable contains the key-value pairs described in Table 4–4.

<table>
<thead>
<tr>
<th>Key (String)</th>
<th>Value Type</th>
<th>Value or Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>Destination name</td>
</tr>
<tr>
<td>type</td>
<td>String</td>
<td>Destination type (queue or topic)</td>
</tr>
<tr>
<td>isTemporary</td>
<td>Boolean</td>
<td>Is destination temporary?</td>
</tr>
</tbody>
</table>

Notice that the destination name and type could be extracted directly from the metrics topic destination name, but the hashtable includes it for your convenience.

By enumerating through the map names and extracting the hashtable described in Table 4–4, you can form a complete list of destination names and some of their characteristics.

The destination list does not include the following kinds of destinations:

- Destinations that are used by Message Queue administration tools
- Destinations that the Message Queue broker creates for internal use

Destination Metrics

The messages you receive when you subscribe to the topic `mq.metrics.destination.queue.monitored_destination_name` have the type property `mq.metrics.destination.queue.monitored_destination_name` set in the message header. The messages you receive when you subscribe to the topic `mq.metrics.destination.topic.monitored_destination_name` have the type property `mq.metrics.destination.topic.monitored_destination_name` set in the message header. Either of these messages has the data listed in Table 4–5 in the message body.
### TABLE 4-5  Data in the Body of a Destination Metrics Message

<table>
<thead>
<tr>
<th>Metric Name</th>
<th>ValueType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numActiveConsumers</td>
<td>long</td>
<td>Current number of active consumers</td>
</tr>
<tr>
<td>avgNumActiveConsumers</td>
<td>long</td>
<td>Average number of active consumers since the broker was last started</td>
</tr>
<tr>
<td>peakNumActiveConsumers</td>
<td>long</td>
<td>Peak number of active consumers since the broker was last started</td>
</tr>
<tr>
<td>numBackupConsumers</td>
<td>long</td>
<td>Current number of backup consumers (applies only to queues)</td>
</tr>
<tr>
<td>avgNumBackupConsumers</td>
<td>long</td>
<td>Average number of backup consumers since the broker was last started (applies only to queues)</td>
</tr>
<tr>
<td>peakNumBackupConsumers</td>
<td>long</td>
<td>Peak number of backup consumers since the broker was last started (applies only to queues)</td>
</tr>
<tr>
<td>numMsgsIn</td>
<td>long</td>
<td>Number of JMS messages that have flowed into this destination since the broker was last started</td>
</tr>
<tr>
<td>numMsgsOut</td>
<td>long</td>
<td>Number of JMS messages that have flowed out of this destination since the broker was last started</td>
</tr>
<tr>
<td>numMsgs</td>
<td>long</td>
<td>Number of JMS messages currently stored in destination memory and persistent store</td>
</tr>
<tr>
<td>avgNumMsgs</td>
<td>long</td>
<td>Average number of JMS messages stored in destination memory and persistent store since the broker was last started</td>
</tr>
<tr>
<td>peakNumMsgs</td>
<td>long</td>
<td>Peak number of JMS messages stored in destination memory and persistent store since the broker was last started</td>
</tr>
<tr>
<td>msgBytesIn</td>
<td>long</td>
<td>Number of JMS message bytes that have flowed into this destination since the broker was last started</td>
</tr>
<tr>
<td>msgBytesOut</td>
<td>long</td>
<td>Number of JMS message bytes that have flowed out of this destination since the broker was last started</td>
</tr>
<tr>
<td>totalMsgBytes</td>
<td>long</td>
<td>Current number of JMS message bytes stored in destination memory and persistent store</td>
</tr>
<tr>
<td>avgTotalMsgBytes</td>
<td>long</td>
<td>Average number of JMS message bytes stored in destination memory and persistent store since the broker was last started</td>
</tr>
<tr>
<td>peakTotalMsgBytes</td>
<td>long</td>
<td>Peak number of JMS message bytes stored in destination memory and persistent store since the broker was last started</td>
</tr>
<tr>
<td>peakMsgBytes</td>
<td>long</td>
<td>Peak number of JMS message bytes in a single message since the broker was last started</td>
</tr>
</tbody>
</table>
### Metrics Monitoring Client Code Examples

Several complete monitoring example applications (including source code and full documentation) are provided when you install Message Queue. You’ll find the examples in your IMQ home directory under `/demo/monitoring`. Before you can run these clients, you must set up your environment (for example, the CLASSPATH environment variable). For details, see "Setting Up Your Environment" on page 17.

Next are brief descriptions of three examples—Broker Metrics, Destination List Metrics, and Destination Metrics—with annotated code examples from each.

These examples use the utility classes MetricsPrinter and MultiColumnPrinter to print formatted and aligned columns of text output. However, rather than explaining how those utility classes are used, the following code examples focus on how to subscribe to the metrics topic and how to extract information from the metrics messages received.

Notice that in the source files, the code for subscribing to metrics topics and processing messages is actually spread across various methods. However, for the sake of clarity, the examples are shown here as though they were contiguous blocks of code.

### A Broker Metrics Example

The source file for this code example is `BrokerMetrics.java`. This metrics monitoring client subscribes to the topic `mq.metrics.broker` and prints broker-related metrics to the standard output.

**Example 4-1** shows how to subscribe to `mq.metrics.broker`.

**EXAMPLE 4-1**    Example of Subscribing to a Broker Metrics Topic

```java
com.sun.messaging.TopicConnectionFactory metricConnectionFactory;
    TopicConnection metricConnection;
    TopicSession metricSession;
```
EXAMPLE 4–1  Example of Subscribing to a Broker Metrics Topic  
(Continued)

```
TopicSubscriber metricSubscriber;
Topic metricTopic;

metricConnectionFactory = new com.sun.messaging.TopicConnectionFactory();
metricConnection = metricConnectionFactory.createTopicConnection();
metricConnection.start();
metricSession = metricConnection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
metricTopic = metricSession.createTopic("mq.metrics.broker");
metricSubscriber = metricSession.createSubscriber(metricTopic);
metricSubscriber.setMessageListener(this);
```

The incoming message is processed in the `onMessage()` and `doTotals()` methods, as shown in Example 4–2.

EXAMPLE 4–2  Example of Processing a Broker Metrics Message

```
public void onMessage(Message m) { 
    try { 
        MapMessage mapMsg = (MapMessage)m;
        String type = mapMsg.getStringProperty("type");
        if (type.equals("mq.metrics.broker")) { 
            if (showTotals) { 
                doTotals(mapMsg); 
                ... 
            }
        }
    }
}

private void doTotals(MapMessage mapMsg) { 
    try { 
        String oneRow[] = new String[ 8 ];
        int i = 0;

        /*
        * Extract broker metrics
        */
        oneRow[i++] = Long.toString(mapMsg.getLong("numMsgsIn"));
        oneRow[i++] = Long.toString(mapMsg.getLong("numMsgsOut"));
        oneRow[i++] = Long.toString(mapMsg.getLong("msgBytesIn"));
```
EXAMPLE 4–2 Example of Processing a Broker Metrics Message (Continued)

```java
    oneRow[i++] = Long.toString(mapMsg.getLong("msgBytesOut"));
    oneRow[i++] = Long.toString(mapMsg.getLong("numPktsIn"));
    oneRow[i++] = Long.toString(mapMsg.getLong("numPktsOut"));
    oneRow[i++] = Long.toString(mapMsg.getLong("pktBytesIn"));
    oneRow[i++] = Long.toString(mapMsg.getLong("pktBytesOut"));
    ...
    } catch (Exception e) {
       System.err.println("onMessage: Exception caught: " + e);
    }
```

Notice how the metric type is extracted, using the `getStringProperty()` method, and is checked. If you use the `onMessage()` method to process metrics messages of different types, you can use the type property to distinguish between different incoming metrics messages.

Also notice how various pieces of information on the broker are extracted, using the `getLong()` method of `mapMsg`.

Run this example monitoring client with the following command:

```
java BrokerMetrics
```

The output looks like the following:

```
<table>
<thead>
<tr>
<th>Msgs</th>
<th>Msg Bytes</th>
<th>Pkts</th>
<th>Pkt Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>Out</td>
<td>In</td>
<td>Out</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>633</td>
<td>7</td>
</tr>
</tbody>
</table>
```

**A Destination List Metrics Example**

The source file for this code example is `DestListMetrics.java`. This client application monitors the list of destinations on a broker by subscribing to the topic `mq.metrics.destination_list`. The messages that arrive contain information describing the destinations that currently exist on the broker, such as destination name, destination type, and whether the destination is temporary.

Example 4–3 shows how to subscribe to `mq.metrics.destination_list`. 
Example 4-3  Example of Subscribing to the Destination List Metrics Topic

```java
com.sun.messaging.TopicConnectionFactory metricConnectionFactory;
TopicConnection metricConnection;
TopicSession metricSession;
TopicSubscriber metricSubscriber;
Topic metricTopic;
String metricTopicName = null;

metricConnectionFactory = new com.sun.messaging.TopicConnectionFactory();
metricConnection = metricConnectionFactory.createTopicConnection();
metricConnection.start();

metricSession = metricConnection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);

metricTopicName = "mq.metrics.destination_list";
metricTopic = metricSession.createTopic(metricTopicName);
metricSubscriber = metricSession.createSubscriber(metricTopic);
metricSubscriber.setMessageListener(this);

The incoming message is processed in the onMessage() method, as shown in Example 4-4:

Example 4-4  Example of Processing a Destination List Metrics Message

```java
public void onMessage(Message m) {
    try{
        MapMessage mapMsg = (MapMessage)m;
        String type = mapMsg.getStringProperty("type");

        if (type.equals(metricTopicName)) {
            String oneRow[3] = new String[3];

            /*
             * Extract metrics
             */
            for (Enumeration e = mapMsg.getMapNames();
                e.hasMoreElements();)
                {
                String metricDestName = (String)e.nextElement();
                Hashtable destValues =
                    (Hashtable)mapMsg.getObject(metricDestName);
                int i = 0;

                oneRow[i++] = (destValues.get("name")).toString();
                oneRow[i++] = (destValues.get("type")).toString();
```
EXAMPLE 4–4  Example of Processing a Destination List Metrics Message  

(Continued)

```java
    oneRow[i++] = (destValues.get("isTemporary")).toString();
    mp.add(oneRow);
}
mp.print();
System.out.println("\n");
mp.clear();
}
else {
    System.err.println("Msg received: not destination list metric type");
}
} catch (Exception e) {
    System.err.println("onMessage: Exception caught: " + e);
}
```

Notice how the metric type is extracted and checked, and how the list of destinations is extracted. By iterating through the map names in `mapMsg` and extracting the corresponding value (a hashtable), you can construct a list of all the destinations and their related information.

As discussed in “Format of Metrics Messages” on page 149, these map names are metrics topic names having one of two forms:

```
mq.metrics.destination.queue.monitored_destination_name
mq.metrics.destination.topic.monitored_destination_name
```

(The map names can also be used to monitor a destination, but that is not done in this particular example.)

Notice that from each extracted hashtable, the information on each destination is extracted using the keys `name`, `type`, and `isTemporary`. The extraction code from the previous code example is reiterated here for your convenience.

EXAMPLE 4–5  Example of Extracting Destination Information From a Hash Table

```java
String metricDestName = (String)e.nextElement();
Hashtable destValues = (Hashtable)mapMsg.getObject(metricDestName);
int i = 0;
    oneRow[i++] = (destValues.get("name")).toString();
    oneRow[i++] = (destValues.get("type")).toString();
    oneRow[i++] = (destValues.get("isTemporary")).toString();
```
EXAMPLE 4–5  Example of Extracting Destination Information From a Hash Table  (Continued)

Run this example monitoring client with the following command:

```
java DestListMetrics
```

The output looks like the following:

<table>
<thead>
<tr>
<th>Destination Name</th>
<th>Type</th>
<th>IsTemporary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SimpleQueue</td>
<td>queue</td>
<td>false</td>
</tr>
<tr>
<td>fooQueue</td>
<td>queue</td>
<td>false</td>
</tr>
<tr>
<td>topic1</td>
<td>topic</td>
<td>false</td>
</tr>
</tbody>
</table>

A Destination Metrics Example

The source file for this code example is `DestMetrics.java`. This client application monitors a specific destination on a broker. It accepts the destination type and name as parameters, and it constructs a metrics topic name of the form

```
mq.metrics.destination.queue. monitored_destination_name or
mq.metrics.destination.topic. monitored_destination_name
```

Example 4–6 shows how to subscribe to the metrics topic for monitoring a specified destination.

EXAMPLE 4–6  Example of Subscribing to a Destination Metrics Topic

```java
com.sun.messaging.TopicConnectionFactory metricConnectionFactory;
TopicConnection metricConnection;
TopicSession metricSession;
TopicSubscriber metricSubscriber;
Topic metricTopic;
String metricTopicName = null;
String destName = null,
destType = null;
for (int i = 0; i < args.length; ++i) {
    ...
    } else if (args[i].equals("-n")) {
        destName = args[i+1];
    } else if (args[i].equals("-t")) {
        destType = args[i+1];
    }
```
EXAMPLE 4–6  Example of Subscribing to a Destination Metrics Topic  (Continued)

```java
metricConnectionFactory = new com.sun.messaging.TopicConnectionFactory();
metricConnection = metricConnectionFactory.createTopicConnection();
metricConnection.start();
metricSession = metricConnection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
if (destType.equals("q")) {
    metricTopicName = "mq.metrics.destination.queue." + destName;
} else {
    metricTopicName = "mq.metrics.destination.topic." + destName;
}
metricTopic = metricSession.createTopic(metricTopicName);
metricSubscriber = metricSession.createSubscriber(metricTopic);
metricSubscriber.setMessageListener(this);
```

The incoming message is processed in the `onMessage()` method, as shown in Example 4–7:

EXAMPLE 4–7  Example of Processing a Destination Metrics Message

```java
public void onMessage(Message m) {
    try {
        MapMessage mapMsg = (MapMessage)m;
        String type = mapMsg.getStringProperty("type");
        if (type.equals(metricTopicName)) {
            String oneRow[] = new String[11];
            int i = 0;
            /*
             * Extract destination metrics
             */
            oneRow[i++] = Long.toString(mapMsg.getLong("numMsgsIn"));
            oneRow[i++] = Long.toString(mapMsg.getLong("numMsgsOut"));
            oneRow[i++] = Long.toString(mapMsg.getLong("msgBytesIn"));
            oneRow[i++] = Long.toString(mapMsg.getLong("msgBytesOut"));
            oneRow[i++] = Long.toString(mapMsg.getLong("numMsgs"));
            oneRow[i++] = Long.toString(mapMsg.getLong("peakNumMsgs"));
            oneRow[i++] = Long.toString(mapMsg.getLong("avgNumMsgs"));
```
EXAMPLE 4-7  Example of Processing a Destination Metrics Message  (Continued)

```java
oneRow[i++] = Long.toString(mapMsg.getLong("totalMsgBytes")/1024);
oneRow[i++] = Long.toString(mapMsg.getLong("peakTotalMsgBytes")/1024);
oneRow[i++] = Long.toString(mapMsg.getLong("avgTotalMsgBytes")/1024);
oneRow[i++] = Long.toString(mapMsg.getLong("peakMsgBytes")/1024);
mp.add(oneRow);
```

```java
} catch (Exception e) {
    System.err.println("onMessage: Exception caught: " + e);
}
```

Notice how the metrics type is extracted, using the `getStringProperty()` method as in the previous examples, and is checked. Also notice how various destination data are extracted, using the `getLong()` method of `mapMsg`.

You can run this example monitoring client with one of the following commands:

```
java DestMetrics -t t -n topic_name
java DestMetrics -t q -n queue_name
```

Using a queue named SimpleQueue as an example, the command would be:

```
java DestMetrics -t q -n SimpleQueue
```

The output looks like the following:

<table>
<thead>
<tr>
<th>Msgs</th>
<th>Msg Bytes</th>
<th>Msg Count</th>
<th>Tot Msg Bytes(k)</th>
<th>Largest Msg</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>Out</td>
<td>In</td>
<td>Out</td>
<td>Curr Peak Avg</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>318000</td>
<td>0</td>
<td>500 250 310 310</td>
</tr>
</tbody>
</table>
SOAP is a protocol that allows for the exchange of data whose structure is defined by an XML scheme. Using Message Queue, you can send JMS messages that contain a SOAP payload. This allows you to transport SOAP messages reliably and to publish SOAP messages to JMS subscribers. This chapter covers the following topics:

- “What is SOAP?” on page 161
- “SOAP Messaging in JAVA” on page 167
- “SOAP Messaging Models and Examples” on page 174
- “Integrating SOAP and Message Queue” on page 185

If you are familiar with the SOAP specification, you can skip the introductory section and start by reading “SOAP Messaging in JAVA” on page 167.

### What is SOAP?

SOAP, the Simple Object Access Protocol, is a protocol that allows the exchange of structured data between peers in a decentralized, distributed environment. The structure of the data being exchanged is specified by an XML scheme.

The fact that SOAP messages are encoded in XML makes SOAP messages portable, because XML is a portable, system-independent way of representing data. By representing data using XML, you can access data from legacy systems as well as share your data with other enterprises. The data integration offered by XML also makes this technology a natural for Web-based computing such as Web services. Firewalls can recognize SOAP packets based on their content type (text/xml-SOAP) and can filter messages based on information exposed in the SOAP message header.

The SOAP specification describes a set of conventions for exchanging XML messages. As such, it forms a natural foundation for Web services that also need to exchange information encoded in XML. Although any two partners could define their own protocol for carrying on this exchange, having a standard such as SOAP allows developers to build the generic pieces that
support this exchange. These pieces might be software that adds functionality to the basic SOAP exchange, or might be tools that administer SOAP messaging, or might even comprise parts of an operating system that supports SOAP processing. Once this support is put in place, other developers can focus on creating the Web services themselves.

The SOAP protocol is fully described at http://www.w3.org/TR/SOAP. This section restricts itself to discussing the reasons why you would use SOAP and to describing basic concepts that will make it easier to work with SOAP messages.

**SOAP with Attachments API for Java**

The Soap with Attachments API for Java (SAAJ) is a JAVA-based API that enforces compliance to the SOAP standard. When you use this API to assemble and disassemble SOAP messages, it ensures the construction of syntactically correct SOAP messages. SAAJ also makes it possible to automate message processing when several applications need to handle different parts of a message before forwarding it to the next recipient.

Figure 5–1 shows the layers that can come into play in the implementation of SOAP messaging. This chapter focuses on the SOAP and language implementation layers.

---

**FIGURE 5–1 SOAP Messaging Layers**

<table>
<thead>
<tr>
<th>Profile (Messaging and Delivery Semantics)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language Implementation</td>
</tr>
<tr>
<td>SOAP with Attachments Encoding</td>
</tr>
<tr>
<td>Wire Transport Protocol</td>
</tr>
</tbody>
</table>

The sections that follow describe each layer shown in the preceding figure in greater detail. The rest of this chapter focuses on the SOAP and language implementation layers.

**The Transport Layer**

Underlying any messaging system is the transport or wire protocol that governs the serialization of the message as it is sent across a wire and the interpretation of the message bits when it gets to the other side. Although SOAP messages can be sent using any number of protocols, the SOAP specification defines only the binding with HTTP. SOAP uses the HTTP request/response message model. It provides SOAP request parameters in an HTTP request and
SOAP response parameters in an HTTP response. The HTTP binding has the advantage of allowing SOAP messages to go through firewalls.

**The SOAP Layer**

Above the transport layer is the SOAP layer. This layer, which is defined in the SOAP Specification, specifies the XML scheme used to identify the message parts: envelope, header, body, and attachments. All SOAP message parts and contents, except for the attachments, are written in XML. The following sample SOAP message shows how XML tags are used to define a SOAP message:

```xml
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
    <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m="Some-URI">
        <symbol>DIS</symbol>
      </m:GetLastTradePrice>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

The wire transport and SOAP layers are actually sufficient to do SOAP messaging. You could create an XML document that defines the message you want to send, and you could write HTTP commands to send the message from one side and to receive it on the other. In this case, the client is limited to sending synchronous messages to a specified URL. Unfortunately, the scope and reliability of this kind of messaging is severely restricted. To overcome these limitations, the *provider* and *profile* layers are added to SOAP messaging.

**The Language Implementation Layer**

A language implementation allows you to create XML messages that conform to SOAP, using API calls. For example, the SAAJ implementation of SOAP allows a Java client to construct a SOAP message and all its parts as Java objects. The client would also use SAAJ to create a connection and use it to send the message. Likewise, a Web service written in Java could use the same implementation (SAAJ), or any other language implementation, to receive the message, to disassemble it, and to acknowledge its receipt.

**The Profiles Layer**

In addition to a language implementation, a SOAP implementation can offer services that relate to message delivery. These could include reliability, persistence, security, and administrative control, and are typically delivered by a SOAP messaging provider. These services will be provided for SOAP messaging by Message Queue in future releases.
Interoperability

Because SOAP providers must all construct and deconstruct messages as defined by the SOAP specification, clients and services using SOAP are interoperable. That is, as shown in Figure 5–2, the client and the service doing SOAP messaging do not need to be written in the same language nor do they need to use the same SOAP provider. It is only the packaging of the message that must be standard.

In order for a SAAJ client or service to interoperate with a service or client using a different implementation, the parties must agree on two things:

- They must use the same transport bindings—that is, the same wire protocol.
- They must use the same profile in constructing the SOAP message being sent.

The SOAP Message

Having surveyed the SOAP messaging layers, let’s examine the SOAP message itself. Although the work of rendering a SOAP message in XML is taken care of by the SAAJ implementation, you must still understand its structure in order to make the SAAJ calls in the right order.

A SOAP message is an XML document that consists of a SOAP envelope, an optional SOAP header, and a SOAP body. The SOAP message header contains information that allows the message to be routed through one or more intermediate nodes before it reaches its final destination.
The envelope is the root element of the XML document representing the message. It defines the framework for how the message should be handled and by whom. Once it encounters the Envelope element, the SOAP processor knows that the XML is a SOAP message and can then look for the individual parts of the message.

The header is a generic mechanism for adding features to a SOAP message. It can contain any number of child elements that define extensions to the base protocol. For example, header child elements might define authentication information, transaction information, locale information, and so on. The actors, the software that handle the message may, without prior agreement, use this mechanism to define who should deal with a feature and whether the feature is mandatory or optional.

The body is a container for mandatory information intended for the ultimate recipient of the message.

A SOAP message may also contain an attachment, which does not have to be in XML. For more information, see “SOAP Packaging Models” on page 165 next.

A SOAP message is constructed like a nested matrioshka doll. When you use SAAJ to assemble or disassemble a message, you need to make the API calls in the appropriate order to get to the message part that interests you. For example, in order to add content to the message, you need to get to the body part of the message. To do this you need to work through the nested layers: SOAP part, SOAP envelope, SOAP body, until you get to the SOAP body element that you will use to specify your data. For more information, see “The SOAP Message Object” on page 168.

**SOAP Packaging Models**

The SOAP specification describes two models of SOAP messages: one that is encoded entirely in XML and one that allows the sender to add an attachment containing non-XML data. You should look over the following two figures and note the parts of the SOAP message for each model. When you use SAAJ to define SOAP messages and their parts, it will be helpful for you to be familiar with this information.

Figure 5–3 shows the SOAP model without attachments. This package includes a SOAP envelope, a header, and a body. The header is optional.
When you construct a SOAP message using SAAJ, you do not have to specify which model you’re following. If you add an attachment, a message like that shown in Figure 5–4 is constructed; if you don’t, a message like that shown in Figure 5–3 is constructed.

Figure 5–3 shows a SOAP Message with attachments. The attachment part can contain any kind of content: image files, plain text, and so on. The sender of a message can choose whether to create a SOAP message with attachments. The message receiver can also choose whether to consume an attachment.

A message that contains one or more attachments is enclosed in a MIME envelope that contains all the parts of the message. In SAAJ, the MIME envelope is automatically produced whenever the client creates an attachment part. If you add an attachment to a message, you are responsible for specifying (in the MIME header) the type of data in the attachment.
The SOAP specification does not provide a programming model or even an API for the construction of SOAP messages; it simply defines the XML schema to be used in packaging a SOAP message.

SAAJ is an application programming interface that can be implemented to support a programming model for SOAP messaging and to furnish Java objects that application or tool writers can use to construct, send, receive, and examine SOAP messages. SAAJ defines two packages:

- javax.xml.soap: you use the objects in this package to define the parts of a SOAP message and to assemble and disassemble SOAP messages. You can also use this package to send a SOAP message without the support of a provider.
- javax.xml.messaging: you use the objects in this package to send a SOAP message using a provider and to receive SOAP messages.

Note – Beginning with SAAJ 1.3, you must put the file mail.jar explicitly in CLASSPATH.

This chapter focuses on the javax.xml.soap package and how you use the objects and methods it defines.
- to assemble and disassemble SOAP messages
- to send and receive these messages

It also explains how you can use the JMS API and Message Queue to send and receive JMS messages that carry SOAP message payloads.

**The SOAP Message Object**

A SOAP Message Object is a tree of objects as shown in **Figure 5–5**. The classes or interfaces from which these objects are derived are all defined in the *javax.xml.soap* package.

As shown in the figure, the SOAPMessage object is a collection of objects divided in two parts: a SOAP part and an attachment part. The main thing to remember is that the attachment part can contain non-xml data.
The SOAP part of the message contains an envelope that contains a body (which can contain data or fault information) and an optional header. When you use SAAJ to create a SOAP message, the SOAP part, envelope, and body are created for you; you need only create the body elements. To do that you need to get to the parent of the body element, the SOAP body.

In order to reach any object in the SOAPMessage tree, you must traverse the tree starting from the root, as shown in the following lines of code. For example, assuming the SOAPMessage is MyMsg, here are the calls you would have to make in order to get the SOAP body:

```java
SOAPPart MyPart = MyMsg.getSOAPPart();
SOAPEnvelope MyEnv = MyPart.getEnvelope();
SOAPBody MyBody = envelope.getBody();
```

At this point, you can create a name for a body element (as described in “Namespaces” on page 171) and add the body element to the SOAPMessage.

For example, the following code line creates a name (a representation of an XML tag) for a body element:

```java
Name bodyName = envelope.createName("Temperature");
```

The next code line adds the body element to the body:

```java
SOAPBodyElement myTemp = MyBody.addBodyElement(bodyName);
```

Finally, this code line defines some data for the body element bodyName:

```java
myTemp.addTextNode("98.6");
```

### Inherited Methods

The elements of a SOAP message form a tree. Each node in that tree implements the Node interface and, starting at the envelope level, each node implements the SOAPElement interface as well. The resulting shared methods are described in Table 5–1.

#### Table 5–1  Inherited Methods

<table>
<thead>
<tr>
<th>Inherited From</th>
<th>Method Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAPElemen</td>
<td>addAttribute(Name, String)</td>
<td>Add an attribute with the specified Name object and string value</td>
</tr>
<tr>
<td></td>
<td>addChildElement(Name)</td>
<td>Create a new SOAPElemen object, initialized with the given Name object, and add the new element</td>
</tr>
<tr>
<td></td>
<td>addChildElement(String, String)</td>
<td>(Use the Envelope.createName method to create a Name object)</td>
</tr>
<tr>
<td>Inherited From</td>
<td>Method Name</td>
<td>Purpose</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>---------</td>
</tr>
<tr>
<td></td>
<td><code>addNamespaceDeclaration(String, String)</code></td>
<td>Add a namespace declaration with the specified prefix and URI</td>
</tr>
<tr>
<td></td>
<td><code>addTextnode(String)</code></td>
<td>Create a new Text object initialized with the given String and add it to this SOAPElement object</td>
</tr>
<tr>
<td></td>
<td><code>getAllAttributes()</code></td>
<td>Return an iterator over all the attribute names in this object</td>
</tr>
<tr>
<td></td>
<td><code>getAttributeValue(Name)</code></td>
<td>Return the value of the specified attribute</td>
</tr>
<tr>
<td></td>
<td><code>getChildElements()</code></td>
<td>Return an iterator over all the immediate content of this element</td>
</tr>
<tr>
<td></td>
<td><code>getChildElements(Name)</code></td>
<td>Return an iterator over all the child elements with the specified name</td>
</tr>
<tr>
<td></td>
<td><code>getElementName()</code></td>
<td>Return the name of this object</td>
</tr>
<tr>
<td></td>
<td><code>getEncodingStyle()</code></td>
<td>Return the encoding style for this object</td>
</tr>
<tr>
<td></td>
<td><code>getNamespacePrefixes()</code></td>
<td>Return an iterator of namespace prefixes</td>
</tr>
<tr>
<td></td>
<td><code>getNamespaceURI(String)</code></td>
<td>Return the URI of the namespace with the given prefix</td>
</tr>
<tr>
<td></td>
<td><code>removeAttribute(Name)</code></td>
<td>Remove the specified attribute</td>
</tr>
<tr>
<td></td>
<td><code>removeNamespaceDeclaration(String)</code></td>
<td>Remove the namespace declaration that corresponds to the specified prefix</td>
</tr>
<tr>
<td></td>
<td><code>setEncodingStyle(String)</code></td>
<td>Set the encoding style for this object to that specified by String</td>
</tr>
<tr>
<td>Node</td>
<td><code>detachNode()</code></td>
<td>Remove this Node object from the tree</td>
</tr>
<tr>
<td></td>
<td><code>getParentElement()</code></td>
<td>Return the parent element of this Node object</td>
</tr>
<tr>
<td></td>
<td><code>getValue</code></td>
<td>Return the value of the immediate child of this Node object if a child exists and its value is text</td>
</tr>
<tr>
<td></td>
<td><code>recycleNode()</code></td>
<td>Notify the implementation that his Node object is no longer being used and is free for reuse</td>
</tr>
<tr>
<td></td>
<td><code>setParentElement(SOAPElement)</code></td>
<td>Set the parent of this object to that specified by the SOAPElement parameter</td>
</tr>
</tbody>
</table>
Namespaces

An XML namespace is a means of qualifying element and attribute names to disambiguate them from other names in the same document. This section provides a brief description of XML namespaces and how they are used in SOAP. For complete information, see http://www.w3.org/TR/REC-xml-names/

An explicit XML namespace declaration takes the following form:

```xml
<prefix:myElement
xmlns:prefix="URI">
```

The declaration defines prefix as an alias for the specified URI. In the element myElement, you can use prefix with any element or attribute to specify that the element or attribute name belongs to the namespace specified by the URI.

The following is an example of a namespace declaration:

```xml
```

This declaration defines SOAP_ENV as an alias for the namespace:

http://schemas.xmlsoap.org/soap/envelope/

After defining the alias, you can use it as a prefix to any attribute or element in the Envelope element. In Example 5–1, the elements <Envelope> and <Body> and the attribute encodingStyle all belong to the SOAP namespace specified by the http://schemas.xmlsoap.org/soap/envelope/URI.

**Example 5–1**  Explicit Namespace Declarations

```xml
<SOAP-ENV:Envelope
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header>
<HeaderA
xmlns="HeaderURI"
SOAP-ENV:mustUnderstand="0">

   The text of the header

</HeaderA>
</SOAP-ENV:Header>
<SOAP-ENV:Body>

```
Note that the URI that defines the namespace does not have to point to an actual location; its purpose is to disambiguate attribute and element names.

**Pre-defined SOAP Namespaces**

SOAP defines two namespaces:

- The SOAP envelope, the root element of a SOAP message, has the following namespace identifier:
  
  
  
  
  "http://schemas.xmlsoap.org/soap/envelope"

- The SOAP serialization, the URI defining SOAP's serialization rules, has the following namespace identifier:
  
  
  
  "http://schemas.xmlsoap.org/soap/encoding"

When you use SAAJ to construct or consume messages, you are responsible for setting or processing namespaces correctly and for discarding messages that have incorrect namespaces.

**Using Namespaces when Creating a SOAP Name**

When you create the body elements or header elements of a SOAP message, you must use the Name object to specify a well-formed name for the element. You obtain a Name object by calling the method SOAPEnvelope.createName.

When you call this method, you can pass a local name as a parameter or you can specify a local name, prefix, and URI. For example, the following line of code defines a name object bodyName.

```java
Name bodyName = MyEnvelope.createName("TradePrice",
    "GetLTP","http://foo.eztrade.com");
```

This would be equivalent to the namespace declaration:

```xml
```

The following code shows how you create a name and associate it with a SOAPBody element. Note the use and placement of the createName method.

```java
SoapBody body = envelope.getBody();//get body from envelope
Name bodyName = envelope.createName("TradePrice", "GetLTP",
    "http://foo.eztrade.com");
SOAPBodyElement gltp = body.addBodyElement(bodyName);
```
**Parsing Name Objects**

For any given Name object, you can use the following Name methods to parse the name:

- getQualifiedName returns "prefix:LocalName", for the given name, this would be `GetLTP:TradePrice`.
- getURI would return "http://foo.eztrade.com".
- getLocalName would return "TradePrice".
- getPrefix would return "GetLTP".

**Destination, Message Factory, and Connection Objects**

SOAP messaging occurs when a SOAP message, produced by a *message factory*, is sent to an *endpoint* by way of a *connection*.

If you are working without a provider, you must do the following:

- Create a `SOAPConnectionFactory` object.
- Create a `SOAPConnection` object.
- Create an Endpoint object that represents the message's destination.
- Create a `MessageFactory` object and use it to create a message.
- Populate the message.
- Send the message.

If you are working with a provider, you must do the following:

- Create a `ProviderConnectionFactory` object.
- Get a `ProviderConnection` object from the provider connection factory.
- Get a `MessageFactory` object from the provider connection and use it to create a message.
- Populate the message.
- Send the message.

The following three sections describe endpoint, message factory, and connection objects in greater detail.

**Endpoint**

An *endpoint* identifies the final destination of a message. An endpoint is defined either by the `Endpoint` class (if you use a provider) or by the `URLEndpoint` class (if you don’t use a provider).
Constructing an Endpoint

You can initialize an endpoint by calling its constructor. The following code uses a constructor to create a URL Endpoint.

```java
myEndpoint = new URLEndpoint("http://somehost/myServlet");
```

Using the Endpoint to Address a Message

To address a message to an endpoint, specify the endpoint as a parameter to the SOAPConnection.call method, which you use to send a SOAP message.

Message Factory

You use a Message Factory to create a SOAP message.

To instantiate a message factory directly, use a statement like the following:

```java
MessageFactory mf = MessageFactory.newInstance();
```

Connection

To send a SOAP message using SAAJ, you must obtain a SOAPConnection. You can also transport a SOAP message using Message Queue; for more information, see "Integrating SOAP and Message Queue" on page 185.

SOAP Connection

A SOAPConnection allows you to send messages directly to a remote party. You can obtain a SOAPConnection object simply by calling the static method SOAPConnectionFactory.newInstance(). Neither reliability nor security are guaranteed over this type of connection.

SOAP Messaging Models and Examples

This section explains how you use SAAJ to send and receive a SOAP message. It is also possible to construct a SOAP message using SAAJ and to send it as the payload of a JMS message. For information, see “Integrating SOAP and Message Queue” on page 185.

SOAP Messaging Programming Models

This section provides a brief summary of the programming models used in SOAP messaging using SAAJ.
A SOAP message is sent to an endpoint by way of a point-to-point connection (implemented by the SOAPConnection class).

You use point-to-point connections to establish a request-reply messaging model. The request-reply model is illustrated in Figure 5–6.

Using this model, the client does the following:

- Creates an endpoint that specifies the URL that will be passed to the SOAPConnection.call method that sends the message.
  
  See “Endpoint” on page 173 for a discussion of the different ways of creating an endpoint.
- Creates a SOAPConnection factory and obtains a SOAP connection.
- Creates a message factory and uses it to create a SOAP message.
- Creates a name for the content of the message and adds the content to the message.
- Uses the SOAPConnection.call method to send the message.

It is assumed that the client will ignore the SOAPMessage object returned by the call method because the only reason this object is returned is to unblock the client.

The SOAP service listening for a request-reply message uses a ReqRespListener object to receive messages.

For a detailed example of a client that does point-to-point messaging, see “Writing a SOAP Client” on page 177.

**Working with Attachments**

If a message contains any data that is not XML, you must add it to the message as an attachment. A message can have any number of attachment parts. Each attachment part can contain anything from plain text to image files.
To create an attachment, you must create a URL object that specifies the location of the file that you want to attach to the SOAP message. You must also create a data handler that will be used to interpret the data in the attachment. Finally, you need to add the attachment to the SOAP message.

To create and add an attachment part to the message, you need to use the JavaBeans Activation Framework (JAF) API. This API allows you to determine the type of an arbitrary piece of data, encapsulate access to it, discover the operations available on it, and activate a bean that can perform these operations. You must include the activation.jar library in your application code in order to work with the JavaBeans Activation Framework.

▼ To Create and Add an Attachment

1. Create a URL object and initialize it to contain the location of the file that you want to attach to the SOAP message.
   
   ```java
   URL url = new URL("http://wombats.com/img.jpg");
   ```

2. Create a data handler and initialize it with a default handler, passing the URL as the location of the data source for the handler.
   
   ```java
   DataHandler dh = new DataHandler(url);
   ```

3. Create an attachment part that is initialized with the data handler containing the URL for the image.
   
   ```java
   AttachmentPart ap1 = message.createAttachmentPart(dh);
   ```

4. Add the attachment part to the SOAP message.
   
   ```java
   myMessage.addAttachmentPart(ap1);
   ```

   After creating the attachment and adding it to the message, you can send the message in the usual way.

   If you are using JMS to send the message, you can use the SOAPMessageIntoJMSMessage conversion utility to convert a SOAP message that has an attachment into a JMS message that you can send to a JMS queue or topic using Message Queue.

### Exception and Fault Handling

A SOAP application can use two error reporting mechanisms: SOAP exceptions and SOAP faults:

- Use a SOAP exception to handle errors that occur on the client side during the generation of the SOAP request or the unmarshalling of the response.
Use a SOAP fault to handle errors that occur on the server side when unmarshalling the request, processing the message, or marshalling the response. In response to such an error, server-side code should create a SOAP message that contains a fault element, rather than a body element, and then it should send that SOAP message back to the originator of the message. If the message receiver is not the ultimate destination for the message, it should identify itself as the soapactor so that the message sender knows where the error occurred. For additional information, see “Handling SOAP Faults” on page 182.

Writing a SOAP Client

The following steps show the calls you have to make to write a SOAP client for point-to-point messaging.

▼ To Write a SOAP Client for Point-to-Point Messaging

1. Get an instance of a SOAPConnectionFactory:
   ```java
   SOAPConnectionFactory myFct = SOAPConnectionFactory.newInstance();
   ```

2. Get a SOAP connection from the SOAPConnectionFactory object:
   ```java
   SOAPConnection myCon = myFct.createConnection();
   ```
   The myCon object that is returned will be used to send the message.

3. Get a MessageFactory object to create a message:
   ```java
   MessageFactory myMsgFct = MessageFactory.newInstance();
   ```

4. Use the message factory to create a message:
   ```java
   SOAPMessage message = myMsgFct.createMessage();
   ```
   The message that is created has all the parts that are shown in Figure 5–7.
At this point, the message has no content. To add content to the message, you need to create a SOAP body element, define a name and content for it, and then add it to the SOAP body.

Remember that to access any part of the message, you need to traverse the tree, calling a `get` method on the parent element to obtain the child. For example, to reach the SOAP body, you start by getting the SOAP part and SOAP envelope:

```java
SOAPPart mySPart = message.getSOAPPart();
SOAPEnvelope myEnvp = mySPart.getEnvelope();
```

5. **Now, you can get the body element from the `myEnvp` object:**

```java
SOAPBody body = myEnvp.getBody();
```

The children that you will add to the body element define the content of the message. (You can add content to the SOAP header in the same way.)

6. **When you add an element to a SOAP body (or header), you must first create a name for it by calling the `envelope.createName` method. This method returns a `Name` object, which you must then pass as a parameter to the method that creates the body element (or the header element).**

```java
Name bodyName = envelope.createName("GetLastTradePrice", "m", "http://eztrade.com")
SOAPBodyElement gltp = body.addBodyElement(bodyName);
```
Now create another body element to add to the gltp element:

```java
Name myContent = envelope.createName("symbol");
SOAPElement mySymbol = gltp.addChildElement(myContent);
```

And now you can define data for the body element `mySymbol`:

```java
mySymbol.addTextNode("SUNW");
```

The resulting SOAP message object is equivalent to this XML scheme:

```xml
<SOAP-ENV:Envelope
   xmlns:SOAPENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:m="http://eztrade.com">
   <m:GetLastTradePrice xmlns:m="http://eztrade.com">
       <symbol>SUNW</symbol>
   </m:GetLastTradePrice>
</SOAP-ENV:Envelope>
```

Every time you send a message or write to it, the message is automatically saved. However if you change a message you have received or one that you have already sent, this would be the point when you would need to update the message by saving all your changes. For example:

```java
message.saveChanges();
```

Before you send the message, you must create a `URL Endpoint` object with the URL of the endpoint to which the message is to be sent. (If you use a profile that adds addressing information to the message header, you do not need to do this.)

```java
URL Endpoint endPt = new URL Endpoint("http://eztrade.com//quotes");
```

Now, you can send the message:

```java
SOAPMessage reply = myCon.call(message, endPt);
```

The reply message (reply) is received on the same connection.

Finally, you need to close the `SOAP Connection` object when it is no longer needed:

```java
myCon.close();
```

## Writing a SOAP Service

A SOAP service represents the final recipient of a SOAP message and should currently be implemented as a servlet. You can write your own servlet or you can extend the JAXMServlet class, which is furnished in the soap.messaging package for your convenience. This section describes the task of writing a SOAP service based on the JAXMServlet class.

Your servlet must implement either the `ReqRespListener` or `OneWayListener` interfaces. The difference between these two is that `ReqRespListener` requires that you return a reply.
Using either of these interfaces, you must implement a method called `onMessage(SOAPMsg)`. **JAXMServlet** will call `onMessage` after receiving a message using the HTTP POST method, which saves you the work of implementing your own `doPost()` method to convert the incoming message into a SOAP message.

**Example 5–2** shows the basic structure of a SOAP service that uses the **JAXMServlet** utility class.

**EXAMPLE 5–2**  Skeleton Message Consumer

```java
public class MyServlet extends JAXMServlet implements ReqRespListener
{
    public SOAPMessage onMessage(SOAP Message msg)
    { //Process message here
    }
}
```

**Example 5–3** shows a simple ping message service:

**EXAMPLE 5–3**  A Simple Ping Message Service

```java
public class SOAPEchoServlet extends JAXMServlet implements ReqRespListener{

    public SOAPMessage onMessage(SOAPMessage mySoapMessage) {
        return mySoapMessage
    }
}
```

**Table 5–2** describes the methods that the JAXM servlet uses. If you were to write your own servlet, you would need to provide methods that performed similar work. In extending **JAXMServlet**, you may need to override the `Init` method and the `setMessageFactory` method; you **must** implement the `onMessage` method.

**TABLE 5–2**  JAXMServlet Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void init (ServletConfig)</td>
<td>Passes the ServletConfig object to its parent's constructor and creates a default messageFactory object. If you want incoming messages to be constructed according to a certain profile, you must call the setMessageFactory method and specify the profile it should use in constructing SOAP messages.</td>
</tr>
</tbody>
</table>
**TABLE 5–2 JAXMServlet Methods (Continued)**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void doPost (HTTPRequest, HTTPResponse)</code></td>
<td>Gets the body of the HTTP request and creates a SOAP message according to the default or specified MessageFactory profile. Calls the <code>onMessage()</code> method of an appropriate listener, passing the SOAP message as a parameter. It is recommended that you do not override this method.</td>
</tr>
<tr>
<td><code>void setMessageFactory (MessageFactory)</code></td>
<td>Sets the <code>MessageFactory</code> object. This is the object used to create the SOAP message that is passed to the <code>onMessage</code> method.</td>
</tr>
<tr>
<td><code>MimeHeaders getHeaders (HTTPRequest)</code></td>
<td>Returns a <code>MimeHeaders</code> object that contains the headers in the given HTTPRequest object.</td>
</tr>
<tr>
<td><code>void putHeaders (MimeHeaders, HTTPResponse)</code></td>
<td>Sets the given HTTPResponse object with the headers in the given <code>MimeHeaders</code> object.</td>
</tr>
<tr>
<td><code>onMessage (SOAPMessage)</code></td>
<td>User-defined method that is called by the servlet when the SOAP message is received. Normally this method needs to disassemble the SOAP message passed to it and to send a reply back to the client (if the servlet implements the ReqRespListener interface).</td>
</tr>
</tbody>
</table>

**Disassembling Messages**

The `onMessage` method needs to disassemble the SOAP message that is passed to it by the servlet and process its contents in an appropriate manner. If there are problems in the processing of the message, the service needs to create a SOAP fault object and send it back to the client as described in “Handling SOAP Faults” on page 182.

Processing the SOAP message may involve working with the headers as well as locating the body elements and dealing with their contents. The following code sample shows how you might disassemble a SOAP message in the body of your `onMessage` method. Basically, you need to use a Document Object Model (DOM) API to parse through the SOAP message.

See [http://xml.coverpages.org/dom.html](http://xml.coverpages.org/dom.html) for more information about the DOM API.

**EXAMPLE 5–4 Processing a SOAP Message**

```java
SOAPEnvelope env = reply.getSOAPPart().getEnvelope();
SOAPBody sb = env.getBody();
// create Name object for XElement that we are searching for
Name ElName = env.createName("XElement");

//Get child elements with the name XElement
Iterator it = sb.getChildElements(ElName);

//Get the first matched child element.
```
EXAMPLE 5–4  Processing a SOAP Message  (Continued)

// We know there is only one.
SOAPBodyElement sbe = (SOAPBodyElement) it.next();

// Get the value for XElement
MyValue = sbe.getValue();
}

Handling Attachments

A SOAP message may have attachments. For sample code that shows you how to create and add an attachment, see "Code Samples" on page 190. For sample code that shows you how to receive and process an attachment, see "Code Samples" on page 190.

In handling attachments, you will need to use the Java Activation Framework API. See http://java.sun.com/products/javabeans/glasgow/jaf.html for more information.

Replying to Messages

In replying to messages, you are simply taking on the client role, now from the server side.

Handling SOAP Faults

Server-side code must use a SOAP fault object to handle errors that occur on the server side when unmarshalling the request, processing the message, or marshalling the response. The SOAPFault interface extends the SOAPBodyElement interface.

SOAP messages have a specific element and format for error reporting on the server side: a SOAP message body can include a SOAP fault element to report errors that happen during the processing of a request. Created on the server side and sent from the server back to the client, the SOAP message containing the SOAPFault object reports any unexpected behavior to the originator of the message.

Within a SOAP message object, the SOAP fault object is a child of the SOAP body, as shown in the figure below. Detail and detail entry objects are only needed if one needs to report that the body of the received message was malformed or contained inappropriate data. In such a case, the detail entry object is used to describe the malformed data.
The SOAP Fault element defines the following four sub-elements:

- **faultcode**
  A code (qualified name) that identifies the error. The code is intended for use by software to provide an algorithmic mechanism for identifying the fault. Predefined fault codes are listed in Table 5–3. This element is required.

- **faultstring**
  A string that describes the fault identified by the fault code. This element is intended to provide an explanation of the error that is understandable to a human. This element is required.
• faultactor
  A URI specifying the source of the fault: the actor that caused the fault along the message path. This element is not required if the message is sent to its final destination without going through any intermediaries. If a fault occurs at an intermediary, then that fault must include a faultactor element.

• detail
  This element carries specific information related to the Body element. It must be present if the contents of the Body element could not be successfully processed. Thus, if this element is missing, the client should infer that the body element was processed. While this element is not required for any error except a malformed payload, you can use it in other cases to supply additional information to the client.

Predefined Fault Codes

The SOAP specification lists four predefined faultcode values. The namespace identifier for these is http://schemas.xmlsoap.org/soap/envelope/.

<table>
<thead>
<tr>
<th>Faultcode Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>VersionMismatch</td>
<td>The processing party found an invalid namespace for the SOAP envelope element; that is, the namespace of the SOAP envelope element was not <a href="http://schemas.xmlsoap.org/soap/envelope/">http://schemas.xmlsoap.org/soap/envelope/</a>.</td>
</tr>
<tr>
<td>MustUnderstand</td>
<td>An immediate child element of the SOAP Header element was either not understood or not appropriately processed by the recipient. This element's mustUnderstand attribute was set to 1 (true).</td>
</tr>
<tr>
<td>Client</td>
<td>The message was incorrectly formed or did not contain the appropriate information. For example, the message did not have the proper authentication or payment information. The client should interpret this code to mean that the message must be changed before it is sent again. If this is the code returned, the SOAPFault object should probably include a detailEntry object that provides additional information about the malformed message.</td>
</tr>
<tr>
<td>Server</td>
<td>The message could not be processed for reasons that are not connected with its content. For example, one of the message handlers could not communicate with another message handler that was upstream and did not respond. Or, the database that the server needed to access is down. The client should interpret this error to mean that the transmission could succeed at a later point in time.</td>
</tr>
</tbody>
</table>

These standard fault codes represent classes of faults. You can extend these by appending a period to the code and adding an additional name. For example, you could define a Server.OutOfMemory code, a Server.Down code, and so forth.
Defining a SOAP Fault

Using SAAJ you can specify the value for faultcode, faultstring, and faultactor using methods of the SOAPFault object. The following code creates a SOAP fault object and sets the faultcode, faultstring, and faultactor attributes:

```java
SOAPFault fault;
reply = factory.createMessage();
envp = reply.getSOAPPart().getEnvelope(true);
someBody = envp.getBody();
fault = someBody.addFault();
fault.setFaultCode("Server");
fault.setFaultString("Some Server Error");
fault.setFaultActor(http://xxx.me.com/list/endpoint.esp/)
reply.saveChanges();
```

The server can return this object in its reply to an incoming SOAP message in case of a server error.

The next code sample shows how to define a detail and detail entry object. Note that you must create a name for the detail entry object.

```java
SOAPFault fault = someBody.addFault();
fault.setFaultCode("Server");
fault.setFaultActor("http://foo.com/uri");
fault.setFaultString ("Unknown error");
Detail myDetail = fault.addDetail();
detail.addDetailEntry(envelope.createName("125detail", "m", "Someuri")).addTextNode("the message cannot contain the string //");
reply.saveChanges();
```

Integrating SOAP and Message Queue

This section explains how you can send, receive, and process a JMS message that contains a SOAP payload.

Message Queue provides a utility to help you send and receive SOAP messages using the JMS API. With the support it provides, you can convert a SOAP message into a JMS message and take advantage of the reliable messaging service offered by Message Queue. You can then convert the message back into a SOAP message on the receiving side and use SAAJ to process it.

To send, receive, and process a JMS message that contains a SOAP payload, you must do the following:

- Import the library com.sun.messaging.xml.MessageTransformer. This is the utility whose methods you will use to convert SOAP messages to JMS messages and vice versa.
Before you transport a SOAP message, you must call the 
MessageTransformer.SOAPMessageIntoJMSMessage method. This method transforms the 
SOAP message into a JMS message. You then send the resulting JMS message as you would a 
normal JMS message. For programming simplicity, it would be best to select a destination 
that is dedicated to receiving SOAP messages. That is, you should create a particular queue 
or topic as a destination for your SOAP message and then send only SOAP messages to this 
destination.

```java
Message myMsg = MessageTransformer.SOAPMessageIntoJMSMessage
    (SOAPMessage, Session);
```

The Session argument specifies the session to be used in producing the Message.

On the receiving side, you get the JMS message containing the SOAP payload as you would a 
normal JMS message. You then call the 
MessageTransformer.SOAPMessageFromJMSMessage utility to extract the SOAP message, 
and then use SAAJ to disassemble the SOAP message and do any further processing. For 
example, to obtain the SOAPMessage make a call like the following:

```java
SOAPMessage myMsg = MessageTransformer.SOAPMessageFromJMSMessage
    (Message, MessageFactory);
```

The MessageFactory argument specifies a message factory that the utility should use to 
construct the SOAPMessage from the given JMS Message.

The following sections offer several use cases and code examples to illustrate this process.

**Example 1: Deferring SOAP Processing**

In the first example, illustrated in Figure 5–9, an incoming SOAP message is received by a 
servlet. After receiving the SOAP message, the servlet MyServlet uses the MessageTransformer 
utility to transform the message into a JMS message, and (reliably) forwards it to an application 
that receives it, turns it back into a SOAP message, and processes the contents of the SOAP 
message.

For information on how the servlet receives the SOAP message, see "Writing a SOAP Service" 
on page 179.
To Transform the SOAP Message into a JMS Message and Send the JMS Message

1. Instantiate a `ConnectionFactory` object and set its attribute values, for example:
   ```java
   QueueConnectionFactory myQConnFact = 
   new com.sun.messaging.QueueConnectionFactory();
   ``

2. Use the `ConnectionFactory` object to create a `Connection` object.
   ```java
   QueueConnection myQConn = 
   myQConnFact.createQueueConnection();
   ``

3. Use the `Connection` object to create a `Session` object.
   ```java
   QueueSession myQSess = myQConn.createQueueSession(false, 
   Session.AUTO_ACKNOWLEDGE);
   ``

4. Instantiate a Message Queue Destination administered object corresponding to a physical destination in the Message Queue message service. In this example, the administered object is `mySOAPQueue` and the physical destination to which it refers is `myPSOAPQ`.
   ```java
   Queue mySOAPQueue = new com.sun.messaging.Queue("myPSOAPQ");
   ``

5. Use the `MessageTransformer` utility, as shown, to transform the SOAP message into a JMS message. For example, given a SOAP message named `MySOAPMsg`,
   ```java
   Message MyJMS = MessageTransformer.SOAPMessageIntoJMSMessage (MySOAPMsg, MyQSess);
   ```
6 Create a QueueSender message producer. This message producer, associated with mySOAPQueue, is used to send messages to the queue destination named myPSOAPQ.
    QueueSender myQueueSender = myQSess.createSender(mySOAPQueue);

7 Send a message to the queue.
    myQueueSender.send(myJMS);

▼ To Receive the JMS Message, Transform it into a SOAP Message, and Process It

1 Instantiate a ConnectionFactory object and set its attribute values.
    QueueConnectionFactory myQConnFact = new
        com.sun.messaging.QueueConnectionFactory();

2 Use the ConnectionFactory object to create a Connection object.
    QueueConnection myQConn = myQConnFact.createQueueConnection();

3 Use the Connection object to create one or more Session objects.
    QueueSession myRQSess = myQConn.createQueueSession(false,
        session.AUTO_ACKNOWLEDGE);

4 Instantiate a Destination object and set its name attribute.
    Queue myRQueue = new com.sun.messaging.Queue("mySOAPQ");

5 Use a Session object and a Destination object to create any needed MessageConsumer objects.
    QueueReceiver myQueueReceiver =
        myRQSess.createReceiver(myRQueue);

6 If needed, instantiate a MessageListener object and register it with a MessageConsumer object.

7 Start the QueueConnection you created in “Example 1: Deferring SOAP Processing” on page 186. Messages for consumption by a client can only be delivered over a connection that has been started.
    myQConn.start();

8 Receive a message from the queue.
    The code below is an example of a synchronous consumption of messages:
    Message myJMS = myQueueReceiver.receive();
9 Use the Message Transformer to convert the JMS message back to a SOAP message.

```java
SOAPMessage MySoap =
    MessageTransformer.SOAPMessageFromJMSMessage
    (myJMS, MyMsgFactory);
```

If you specify null for the MessageFactory argument, the default Message Factory is used to construct the SOAP Message.

10 Disassemble the SOAP message in preparation for further processing. See “The SOAP Message Object” on page 168 for information.

Example 2: Publishing SOAP Messages

In the next example, illustrated in Figure 5–10, an incoming SOAP message is received by a servlet. The servlet packages the SOAP message as a JMS message and (reliably) forwards it to a topic. Each application that subscribes to this topic, receives the JMS message, turns it back into a SOAP message, and processes its contents.
The code that accomplishes this is exactly the same as in the previous example, except that instead of sending the JMS message to a queue, you send it to a topic. For an example of publishing a SOAP message using Message Queue, see Example 5–5.

**Code Samples**

This section includes and describes two code samples: one that sends a JMS message with a SOAP payload, and another that receives the JMS/SOAP message and processes the SOAP message.

Example 5–5 illustrates the use of the JMS API, the SAAJ API, and the JAF API to send a SOAP message with attachments as the payload to a JMS message. The code shown for the SendSOAPMessageWithJMS includes the following methods:

- A constructor that calls the init method to initialize all the JMS objects required to publish a message
- A send method that creates the SOAP message and an attachment, converts the SOAP message into a JMS message, and publishes the JMS message
- A close method that closes the connection
- A main method that calls the send and close methods

**EXAMPLE 5–5** Sending a JMS Message with a SOAP Payload

```java
//Libraries needed to build SOAP message
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.SOAPPart;
import javax.xml.soap.SOAPEnvelope;
import javax.xml.soap.SOAPBody;
import javax.xml.soap.SOAPElement;
import javax.xml.soap.MessageFactory;
import javax.xml.soap.AttachmentPart;
import javax.xml.soap.Name

//Libraries needed to work with attachments (Java Activation Framework API)
import java.net.URL;
import javax.activation.DataHandler;

//Libraries needed to convert the SOAP message to a JMS message and to send it
import com.sun.messaging.xml.MessageTransformer;
import com.sun.messaging.BasicConnectionFactory;

//Libraries needed to set up a JMS connection and to send a message
import javax.jms.TopicConnectionFactory;
import javax.jms.TopicConnection;
import javax.jms.JMSEXception;
import javax.jms.Session;
import javax.jms.Message;
import javax.jms.Topic;
import javax.jms.TopicPublisher;

//Define class that sends JMS message with SOAP payload
public class SendSOAPMessageWithJMS{
    TopicConnectionFactory tcf = null;
    TopicConnection tc = null;
    TopicSession session = null;
    Topic topic = null;
    TopicPublisher publisher = null;

    //default constructor method
    public SendSOAPMessageWithJMS(String topicName){
        init(topicName);
    }
```
Sending a JMS Message with a SOAP Payload

```java
// Method to initialize JMS Connection, Session, Topic, and Publisher
public void init(String topicName) {
    try {
        tcf = new com.sun.messaging.TopicConnectionFactory();
        tc = tcf.createTopicConnection();
        session = tc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
        topic = session.createTopic(topicName);
        publisher = session.createPublisher(topic);
    }

    // Method to create and send the SOAP/JMS message
    public void send() throws Exception{
        MessageFactory mf = MessageFactory.newInstance(); // create default factory
        SOAPMessage soapMessage = mf.createMessage(); // create SOAP message object
        SOAPPart soapPart = soapMessage.getSOAPPart(); // start to drill down to body
        SOAPEnvelope soapEnvelope = soapPart.getEnvelope(); // first the envelope
        SOAPBody soapBody = soapEnvelope.getBody();
        Name myName = soapEnvelope.createName("HelloWorld", "hw",
                                                http://www.sun.com/imq");
                      // name for body element
        SOAPElement element = soapBody.addChildElement(myName); // add body element
        element.addTextNode("Welcome to SunOne Web Services."); // add text value

        // Create an attachment with the Java Framework Activation API
        URL url = new URL("http://java.sun.com/webservices/");
        DataHandler dh = new DataHandler(url);
        AttachmentPart ap = soapMessage.createAttachmentPart(dh);

        // Set content type and ID
        ap.setContentType("text/html");
        ap.setContentID("cid-001");

        // Add attachment to the SOAP message
        soapMessage.addAttachmentPart(ap);
        soapMessage.saveChanges();

        // Convert SOAP to JMS message.
        Message m = MessageTransformer.SOAPMessageIntoJMSMessage
                        (soapMessage, session);

        // Publish JMS message
        publisher.publish(m);

        // Close JMS connection
    }
```
EXAMPLE 5–5 Sending a JMS Message with a SOAP Payload  
(Continued)

```java
public void close() throws JMSException {
    tc.close();
}

// Main program to send SOAP message with JMS
public static void main (String[] args) {
    try {
        String topicName = System.getProperty("TopicName");
        if(topicName == null) {
            topicName = "test";
        }

        SendSOAPMEssageWithJMS ssm = new SendSOAPMEssageWithJMS(topicName);
        ssm.send();
        ssm.close();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

Example 5–6 illustrates the use of the JMS API, SAAJ, and the DOM API to receive a SOAP message with attachments as the payload to a JMS message. The code shown for the ReceiveSOAPMessageWithJMS includes the following methods:

- A constructor that calls the init method to initialize all the JMS objects needed to receive a message.
- An onMessage method that delivers the message and which is called by the listener. The onMessage method also calls the message transformer utility to convert the JMS message into a SOAP message and then uses SAAJ to process the SOAP body and uses SAAJ and the DOM API to process the message attachments.
- A main method that initializes the ReceiveSOAPMessageWithJMS class.

EXAMPLE 5–6 Receiving a JMS Message with a SOAP Payload

// Libraries that support SOAP processing
import javax.xml.soap.MessageFactory;
import javax.xml.soap.SOAPMessage;
import javax.xml.soap.AttachmentPart

// Library containing the JMS to SOAP transformer
import com.sun.messaging.xml.MessageTransformer;
EXAMPLE 5-6 Receiving a JMS Message with a SOAP Payload  (Continued)

//Libraries for JMS messaging support
import com.sun.messaging.TopicConnectionFactory

//Interfaces for JMS messaging
import javax.jms.MessageListener;
import javax.jms.TopicConnection;
import javax.jms.TopicSession;
import javax.jms.Message;
import javax.jms.Session;
import javax.jms.Topic;
import javax.jms.JMSException;
import javax.jms.TopicSubscriber

//Library to support parsing attachment part (from DOM API)
import java.util.iterator;

public class ReceiveSOAPMessageWithJMS implements MessageListener{
    TopicConnectionFactory tcf = null;
    TopicConnection tc = null;
    TopicSession session = null;
    Topic topic = null;
    TopicSubscriber subscriber = null;
    MessageFactory messageFactory = null;

    //Default constructor
    public ReceiveSOAPMessageWithJMS(String topicName) {
        init(topicName);
    }

    //Set up JMS connection and related objects
    public void init(String topicName){
        try {
            //Construct default SOAP message factory
            messageFactory = MessageFactory.newInstance();

            //JMS set up
            tcf = new com.sun.messaging.TopicConnectionFactory();
            tc = tcf.createTopicConnectionFactory();
            session = tc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
            topic = session.createTopic(topicName);
            subscriber = session.createSubscriber(topic);
            subscriber.setMessageListener(this);
            tc.start();

            System.out.println("ready to receive SOAP messages...");
        } catch (Exception jmse){
            jmse.printStackTrace();
        }
    }
}
EXAMPLE 5–6  Receiving a JMS Message with a SOAP Payload  (Continued)

    //JMS messages are delivered to the onMessage method
    public void onMessage(Message message){
        try {
            //Convert JMS to SOAP message
            SOAPMessage soapMessage = MessageTransformer.SOAPMessageFromJMSMessage
                (message, messageFactory);

            //Print attachment counts
            System.out.println("message received! Attachment counts:
                * + soapMessage.countAttachments()");

            //Get attachment parts of the SOAP message
            Iterator iterator = soapMessage.getAttachments();
            while (iterator.hasNext()) {
                //Get next attachment
                AttachmentPart ap = (AttachmentPart) iterator.next();

                //Get content type
                String contentType = ap.getContentType();
                System.out.println("content type:
                    + content");

                //Get content id
                String contentID = ap.getContentID();
                System.out.println("content Id:
                    + contentId");

                //Check to see if this is text
                if(contentType.indexOf("text")>=0 {
                    //Get and print string content if it is a text attachment
                    String content = (String) ap.getContent();
                    System.out.println("*** attachment content: " + content);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    //Main method to start sample receiver
    public static void main (String[] args){
        try {
            String topicName = System.getProperty("TopicName");
            if( topicName == null) {

Chapter 5 • Working with SOAP Messages  195
EXAMPLE 5–6  Receiving a JMS Message with a SOAP Payload  (Continued)

    topicName = "test";
    }
    ReceiveSOAPMessageWithJMS rsm = new ReceiveSOAPMessageWithJMS(topicName);
} catch (Exception e) {
    e.printStackTrace();
    }
}
Embedding a Message Queue Broker in a Java Client

Message Queue supports running a broker from within a Java client. Such a broker, called an *embedded broker*, runs in the same JVM as the Java client that creates and starts it.

Beyond operating like a normal standalone broker, an embedded broker offers the application in which it is embedded access to a special kind of connection called a *direct mode connection*. Direct mode connections are used just like ordinary connections, but they are much higher performing because they use in-memory transport instead of TCP. To specify a direct mode connection, the client specifies `mq://localhost/direct` as the broker address in the connection factory from which it subsequently creates the connection.

The following sections provide more information about creating and managing embedded brokers:

- “Creating, Initializing and Starting an Embedded Broker” on page 197
- “Creating a Direct Connection to an Embedded Broker” on page 200
- “Creating a TCP Connection to an Embedded Broker” on page 200
- “Stopping and Shutting Down an Embedded Broker” on page 200
- “Embedded Broker Example” on page 200

**Creating, Initializing and Starting an Embedded Broker**

To create, initialize, and start an embedded broker, you:

1. Create a broker instance in the client runtime.
2. Create a broker event listener.
3. Define properties to use when initializing the broker instance.
4. Initialize the broker instance.
5. Start the broker instance.

The following listing shows an example of creating, initializing, and starting an Embedded Broker. In this example, `args` represents the string of arguments to pass as properties when
initializing the broker instance, and *EmbeddedBrokerEventListener* is an existing class that implements the BrokerEventListener interface.

```java
import com.sun.messaging.jmq.jmsclient.runtime.BrokerInstance;
import com.sun.messaging.jmq.jmsclient.runtime.ClientRuntime;
import com.sun.messaging.jmq.jmsservice.BrokerEvent;
import com.sun.messaging.jmq.jmsservice.BrokerEventListener;

// Obtain the ClientRuntime singleton object
ClientRuntime clientRuntime = ClientRuntime.getRuntime();

// Create a broker instance
BrokerInstance brokerInstance = clientRuntime.createBrokerInstance();

// Create a broker event listener
BrokerEventListener listener = new EmbeddedBrokerEventListener();

// Convert the broker arguments into Properties. Note that parseArgs is
// a utility method that does not change the broker instance.
Properties props = brokerInstance.parseArgs(args);

// Initialize the broker instance using the specified properties and
// broker event listener
brokerInstance.init(props, listener);

// now start the embedded broker
brokerInstance.start();
```

### Creating a Broker Event Listener

When initializing an embedded broker, you must provide a broker event listener. This listener is an instance of a class that implements the BrokerEventListener interface. This interface specifies two methods:

- **public void brokerEvent(BrokerEvent brokerEvent)**, which is called when the broker starts and stops. This method is not required to perform any specific actions, so you can implement an empty method.
- **public boolean exitRequested(BrokerEvent event, Throwable thr)**, which is called when a fatal error has been encountered and the broker is seeking permission to terminate. This method should return true. If it returns false, the broker will not shut down, but it may then cease to respond correctly.

The following listing shows an example class that implements the BrokerEventListener interface.
class EmbeddedBrokerEventListener implements BrokerEventListener {

    public void brokerEvent(BrokerEvent brokerEvent) {
        System.out.println ("Received broker event:"+brokerEvent);
    }

    public boolean exitRequested(BrokerEvent event, Throwable thr) {
        System.out.println ("Broker is requesting a shutdown because of:"+event+" with "+thr);
        // return true to allow the broker to shutdown
        return true;
    }
}

Arguments to Specify WhenInitializing an Embedded Broker

When initializing an embedded broker, you can provide a list of arguments as properties.

Because a Java client runtime (not the imqbrokerd utility) is initializing the broker, you should specify these arguments:

- `-imqhome path` The home directory of the Message Queue installation (see Table P-5)
- `-libhome path` The directory in which Message Queue libraries are stored. This location depends on the distribution from which Message Queue was installed:
  - From an IPS image: IMQ_HOME/ Lib
  - From Solaris SVR4 packages: /usr/share/lib/imq
  - From Linux RPMs: /opt/sun/mq/share/lib
- `-varhome path` The directory in which Message Queue temporary or dynamically created configuration and data files are stored installation (see Table P-5)

You can also specify imqbrokerd options as arguments. Two useful options to specify as arguments are:

- `-name instanceName` The instance name of the broker.
- `-port portNumber` The port number for the broker’s Port Mapper. This is port number on which the broker listens for client connections.
Creating a Direct Connection to an Embedded Broker

Once an embedded broker has been started, you can create direct connections to it from the client in which it is embedded. To do so, you create a connection as you would with an ordinary broker, but you specify `mq://localhost/direct` as broker address in the connection factory. For example:

```java
cf.setProperty(ConnectionConfiguration.imqAddressList, "mq://localhost/direct");
Connection connection = cf.createConnection();
```

Creating a TCP Connection to an Embedded Broker

Once an embedded broker has been started, clients other than the one in which it is embedded can connect to it as though it were an ordinary standalone broker. For example:

```java
cf.setProperty(ConnectionConfiguration.imqAddressList, "mq://myhost.example.com:7676");
Connection connection = cf.createConnection();
```

Stopping and Shutting Down an Embedded Broker

To stop and shut down an embedded broker, use the stop() and shutdown() methods of the broker instance. For example:

```java
// Stop the embedded broker
brokerInstance.stop();
// Shut down the embedded broker
brokerInstance.shutdown();
```

Embedded Broker Example

The following listing demonstrates how to:
- Create, initialize and start an embedded broker
- Create a direct connection
- Send and receive messages across a direct connection
- Stop and shut down an embedded broker
- Create a broker event listener

```java
package test.direct;

import java.util.Properties;
```
public class EmbeddedBrokerExample {
    public void run(String[] args) throws Exception{
        // obtain the ClientRuntime singleton object
        ClientRuntime clientRuntime = ClientRuntime.getRuntime();

        // create the embedded broker instance
        BrokerInstance brokerInstance = clientRuntime.createBrokerInstance();

        // convert the specified broker arguments into Properties
        // this is a utility function: it doesn’t change the broker
        Properties props = brokerInstance.parseArgs(args);

        // initialise the broker instance
        // using the specified properties
        // and a BrokerEventListener
        BrokerEventListener listener = new ExampleBrokerEventListener();
        brokerInstance.init(props, listener);

        // now start the embedded broker
        brokerInstance.start();

        System.out.println("Embedded broker started");

        // now create a direct connection to the embedded broker
        // this is identical to a normal TCP connection except that a special URL is used
        qcf.setProperty(ConnectionConfiguration.imqAddressList, "mq://localhost/direct");

        Connection connection = qcf.createConnection();
        System.out.println("Created direct connection to embedded broker");
// now create a session and a producer and consumer in the normal way
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
Queue queue = session.createQueue("exampleQueue");
MessageConsumer consumer = session.createConsumer(queue);
MessageProducer producer = session.createProducer(queue);

// send a message to the queue in the normal way
TextMessage textMessage = session.createTextMessage("This is a message");
producer.send(textMessage);

// receive a message from the queue in the normal way
connection.start();
Message receivedMessage = consumer.receive(1000);
System.out.println("Received message "+((TextMessage)receivedMessage).getText());

// close the client connection
connection.close();

// stop the embedded broker
brokerInstance.stop();

// shutdown the embedded broker
brokerInstance.shutdown();

}

public static void main(String[] args) throws Exception {
    EmbeddedBrokerExample ebe = new EmbeddedBrokerExample();
ebe.run(args);

}

class ExampleBrokerEventListener implements BrokerEventListener {
    public void brokerEvent(BrokerEvent brokerEvent) {
        System.out.println("Received broker event:"+brokerEvent);}

    public boolean exitRequested(BrokerEvent event, Throwable thr) {
        System.out.println("Broker is requesting a shutdown because of:"+event+" with "+thr);
        // return true to allow the broker to shutdown
        return true;
    }

}
Warning Messages and Client Error Codes

This appendix provides reference information for warning messages and for error codes returned by the Message Queue client runtime when it raises a JMS exception.

- A **warning message** is a message output when the Message Queue Java client runtime experiences a problem that should not occur under normal operating conditions. The message is displayed where the application displays its output. Usually, this is the window from which the application is started. Table A–1 lists Message Queue warning messages.

  In general, a warning message does not cause message loss or affect reliability issues. But when warning messages appear constantly on the application's console, the user should contact Message Queue technical support to diagnose the cause of the warning messages.

- **Error codes** and messages are returned by the client runtime when it raises an exception. You can obtain the error code and its corresponding message using the `JMSException.getErrorCode()` method and the `JMSException.getMessage()` method. Table A–2 lists Message Queue error codes.

Note that warning messages and error codes are not defined in the JMS specification, but are specific to each JMS provider. Applications that rely on these error codes in their programming logic are not portable across JMS providers.
# Warning Messages and Error Codes

## Table A–1 Message Queue Warning Message Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Message and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2000</td>
<td><strong>Message</strong> Warning: Received unknown packet: <em>mq-packet-dump.</em></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The Message Queue client runtime received an unrecognized Message Queue packet, where <em>mq-packet-dump</em> is replaced with the specific Message Queue packet dump that caused this warning message. The Message Queue broker may not be fully compatible with the client runtime version.</td>
</tr>
<tr>
<td>W2001</td>
<td><strong>Message</strong> Warning: pkt not processed, no message consumer:<em>mq-packet-dump.</em></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The Message Queue client runtime received an unexpected Message Queue acknowledge message. The variable <em>mq-packet-dump</em> is replaced with the specific Message Queue packet dump that caused this warning message.</td>
</tr>
<tr>
<td>W2003</td>
<td><strong>Message</strong> Warning: Broker not responding <em>X</em> for <em>Y</em> seconds. Still trying....</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The Message Queue client runtime has not received a response from the broker for more than 2 minutes (default). In the actual message, the <em>X</em> variable is replaced with the Message Queue packet type that the client runtime is waiting for, and the <em>Y</em> variable is replaced with the number of seconds that the client runtime has been waiting for the packet.</td>
</tr>
</tbody>
</table>

Table A–2 lists the error codes in numerical order. For each code listed, it supplies the error message and a probable cause.

Each error message returned has the following format:

```
[Code]: "Message -cause Root-cause-exception-message ."
```

Message text provided for -cause is only appended to the message if there is an exception linked to the JMS exception. For example, a JMS exception with error code C4003 returns the following error message:

```
[C4003]: Error occurred on connection creation [localhost:7676] - cause: java.net.ConnectException: Connection refused: connect
```
### Table A-2 Message Queue Client Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Message Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C4000</td>
<td><strong>Message</strong> Packet acknowledge failed.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was not able to receive or process the expected</td>
</tr>
<tr>
<td></td>
<td>acknowledgment sent from the broker.</td>
</tr>
<tr>
<td>C4001</td>
<td><strong>Message</strong> Write packet failed.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was not able to send information to the broker. This</td>
</tr>
<tr>
<td></td>
<td>might be caused by an underlying network I/O failure or by the JMS connection</td>
</tr>
<tr>
<td></td>
<td>being closed.</td>
</tr>
<tr>
<td>C4002</td>
<td><strong>Message</strong> Read packet failed.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was not able to process inbound message properly. This</td>
</tr>
<tr>
<td></td>
<td>might be caused by an underlying network I/O failure.</td>
</tr>
<tr>
<td>C4003</td>
<td><strong>Message</strong> Error occurred on connection creation [host, port].</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was not able to establish a connection to the broker</td>
</tr>
<tr>
<td></td>
<td>with the specified host name and port number.</td>
</tr>
<tr>
<td>C4004</td>
<td><strong>Message</strong> An error occurred on connection close.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime encountered one or more errors when closing the connection</td>
</tr>
<tr>
<td></td>
<td>to the broker.</td>
</tr>
<tr>
<td>C4005</td>
<td><strong>Message</strong> Get properties from packet failed.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was not able to retrieve a property object from the</td>
</tr>
<tr>
<td></td>
<td>Message Queue packet.</td>
</tr>
<tr>
<td>C4006</td>
<td><strong>Message</strong> Set properties to packet failed.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was not able to set a property object in the Message</td>
</tr>
<tr>
<td></td>
<td>Queue packet.</td>
</tr>
<tr>
<td>C4007</td>
<td><strong>Message</strong> Durable subscription {0} in use.</td>
</tr>
<tr>
<td></td>
<td><strong>0</strong> is replaced with the subscribed destination name.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was not able to unsubscribe the durable subscriber</td>
</tr>
<tr>
<td></td>
<td>because it is currently in use by another consumer.</td>
</tr>
<tr>
<td>C4008</td>
<td><strong>Message</strong> Message in read-only mode.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to write to a JMS Message that is in read-only mode.</td>
</tr>
<tr>
<td>C4009</td>
<td><strong>Message</strong> Message in write-only mode.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to read a JMS Message that is in write-only mode.</td>
</tr>
<tr>
<td>C4010</td>
<td><strong>Message</strong> Read message failed.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was not able to read the stream of bytes from a Bytes</td>
</tr>
<tr>
<td></td>
<td>Message type message.</td>
</tr>
<tr>
<td>Code</td>
<td>Message and Description</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>C4011</td>
<td><strong>Message</strong> Write message failed.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was not able to write the stream of bytes to a BytesMessage type message.</td>
</tr>
<tr>
<td>C4012</td>
<td><strong>Message</strong> message failed.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime encountered an error when processing the <code>reset()</code> method for a BytesMessage or StreamMessage type message.</td>
</tr>
<tr>
<td>C4013</td>
<td><strong>Message</strong> Unexpected end of stream when reading message.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime reached end-of-stream when processing the <code>readXXX()</code> method for a BytesMessage or StreamMessage type message.</td>
</tr>
<tr>
<td>C4014</td>
<td><strong>Message</strong> Serialize message failed.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime encountered an error when processing the serialization of an object, such as <code>ObjectMessage.writeObject(java.io.Serializable object)</code>.</td>
</tr>
<tr>
<td>C4015</td>
<td><strong>Message</strong> Deserialize message failed.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime encountered an error when processing the deserialization of an object, for example, when processing the method <code>ObjectMessage.getObject()</code>.</td>
</tr>
<tr>
<td>C4016</td>
<td><strong>Message</strong> Error occurred during message acknowledgment.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime encountered an error during the process of message acknowledgment in a session.</td>
</tr>
<tr>
<td>C4017</td>
<td><strong>Message</strong> Invalid message format.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime encountered an error when processing a JMS Message; for example, during data type conversion.</td>
</tr>
<tr>
<td>C4018</td>
<td><strong>Message</strong> Error occurred on request message redeliver.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime encountered an error when processing <code>recover()</code> or <code>rollback()</code> for the JMS session.</td>
</tr>
<tr>
<td>C4019</td>
<td><strong>Message</strong> Destination not found: [0].</td>
</tr>
<tr>
<td></td>
<td>`[0] is replaced with the destination name specified in the API parameter.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was unable to process the API request due to an invalid destination specified in the API, for example, the call <code>MessageProducer.send (null, message)</code> raises JMSException with this error code and message.</td>
</tr>
</tbody>
</table>
### TABLE A–2  Message Queue Client Error Codes  (Continued)

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Cause</th>
</tr>
</thead>
</table>
| C4020 | Message Temporary destination belongs to a closed connection or another connection - \[0\].
   \[0\] is replaced with the temporary destination name specified in the API parameter. | An attempt was made to use a temporary destination that is not valid for the message producer. |
<p>| C4021 | Message Consumer not found.                                              | The Message Queue session could not find the message consumer for a message sent from the broker. The message consumer may have been closed by the application or by the client runtime before the message for the consumer was processed. |
| C4022 | Message Selector invalid: [0]. [0] is replaced with the selector string specified in the API parameter. | The client runtime was unable to process the JMS API call because the specified selector is invalid. |
| C4023 | Message Client unacknowledged messages over system defined limit.       | The client runtime raises a JMSException with this error code and message if unacknowledged messages exceed the system defined limit in a CLIENT_ACKNOWLEDGE session. |
| C4024 | Message The session is not transacted.                                  | An attempt was made to use a transacted session API in a non-transacted session. For example, calling the methods commit() or rollback in a AUTO_ACKNOWLEDGE session. |
| C4025 | Message Cannot call this method from a transacted session.              | An attempt was made to call the Session, recover() method from a transacted session. |
| C4026 | Message Client non-committed messages over system defined limit.        | The client runtime raises a JMSException with this error code and message if non committed messages exceed the system-defined limit in a transacted session. |
| C4027 | Message Invalid transaction ID: [0]. [0] is replaced with the internal transaction ID. | An attempt was made to commit or rollback a transacted session with a transaction ID that is no longer valid. |</p>
<table>
<thead>
<tr>
<th>Code</th>
<th>Message and Description</th>
</tr>
</thead>
</table>
| C4028 | **Message** Transaction ID {0} in use.  
(0) is replaced with the internal transaction ID.  
**Cause** The internal transaction ID is already in use by the system. An application should not receive a JMSException with this error code under normal operations. |
| C4029 | **Message** Invalid session for ServerSession.  
**Cause** An attempt was made to use an invalid JMS session for the ServerSession object, for example, no message listener was set for the session. |
| C4030 | **Message** Illegal maxMessages value for ServerSession: {0}.  
(0) was replaced with maxMessages value used by the application.  
**Cause** The configured maxMessages value for ServerSession is less than 0. |
| C4031 | **Message** MessageConsumer and ServerSession session conflict.  
**Cause** An attempt was made to create a message consumer for a session already used by a ServerSession object. |
| C4032 | **Message** Can not use receive() when message listener was set.  
**Cause** An attempt was made to do a synchronous receive with an asynchronous message consumer. |
| C4033 | **Message** Authentication type does not match: {0} and {1}.  
(0) is replaced with the authentication type used by the client runtime.  
(1) is replaced with the authentication type requested by the broker.  
**Cause** The authentication type requested by the broker does not match the authentication type in use by the client runtime. |
| C4034 | **Message** Illegal authentication state.  
**Cause** The authentication handshake failed between the client runtime and the broker. |
| C4035 | **Message** Received AUTHENTICATE_REQUEST status code FORBIDDEN.  
**Cause** The client runtime authentication to the broker failed. |
| C4036 | **Message** A broker error occurred.  
**Cause** A generic error code indicating that the client's requested operation to the broker failed. |
| C4037 | **Message** Broker unavailable or broker timeout.  
**Cause** The client runtime was unable to establish a connection to the broker. |
<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C4038</td>
<td><strong>Message</strong> [4038] - cause: [0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0) is replaced with a root cause exception message.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime caught an exception thrown from the JVM. The client runtime throws JMSException with the &quot;root cause exception&quot; set as the linked exception.</td>
<td></td>
</tr>
<tr>
<td>C4039</td>
<td><strong>Message</strong> Cannot delete destination.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime was unable to delete the specified temporary destination. See TemporaryTopic.delete() and TemporaryQueue.delete() API Javadoc for constraints on deleting a temporary destination.</td>
<td></td>
</tr>
<tr>
<td>C4040</td>
<td><strong>Message</strong> Invalid ObjectProperty type.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to set a non-primitive Java object as a JMS message property. Please see Message.setObjectProperty() API Javadoc for valid object property types.</td>
<td></td>
</tr>
<tr>
<td>C4041</td>
<td><strong>Message</strong> Reserved word used as property name - [0].</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0) is replaced with the property name.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to use a reserved word, defined in the JMS Message API Javadoc, as the message property name, for example, NULL, TRUE, FALSE.</td>
<td></td>
</tr>
<tr>
<td>C4042</td>
<td><strong>Message</strong> Illegal first character of property name - [0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0) is replaced with the illegal character.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to use a property name with an illegal first character. See JMS Message API Javadoc for valid property names.</td>
<td></td>
</tr>
<tr>
<td>C4043</td>
<td><strong>Message</strong> Illegal character used in property name - [0]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0) is replaced with the illegal character used.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to use a property name containing an illegal character. See JMS Message API Javadoc for valid property names.</td>
<td></td>
</tr>
<tr>
<td>C4044</td>
<td><strong>Message</strong> Browser timeout.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The queue browser was unable to return the next available message to the application within the system's predefined timeout period.</td>
<td></td>
</tr>
<tr>
<td>C4045</td>
<td><strong>Message</strong> No more elements.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> In QueueBrowser, the enumeration object has reached the end of element but nextElement() is called by the application.</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Message and Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>C4046</td>
<td><strong>Message</strong> Browser closed.                                                                                                                                         <strong>Cause</strong> An attempt was made to use QueueBrowser methods on a closed QueueBrowser object.</td>
<td></td>
</tr>
<tr>
<td>C4047</td>
<td><strong>Message</strong> Operation interrupted.                                                                                                                                   <strong>Cause</strong> ServerSession was interrupted. The client runtime throws RuntimeException with the above exception message when it is interrupted in the ServerSession.</td>
<td></td>
</tr>
<tr>
<td>C4048</td>
<td><strong>Message</strong> ServerSession is in progress.                                                                                                                           <strong>Cause</strong> Multiple threads attempted to operate on a server session concurrently.</td>
<td></td>
</tr>
<tr>
<td>C4049</td>
<td><strong>Message</strong> Can not call Connection.close(), stop(), etc from message listener.                                                                                       <strong>Cause</strong> An attempt was made to call Connection.close(),...stop(),etc from a message listener.</td>
<td></td>
</tr>
<tr>
<td>C4050</td>
<td><strong>Message</strong> Invalid destination name - {0}.                                                                                                                          <strong>Cause</strong> An attempt was made to use an invalid destination name, for example, NULL.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with the invalid destination name used</td>
<td></td>
</tr>
<tr>
<td>C4051</td>
<td><strong>Message</strong> Invalid delivery parameter. {0} : {1}                                                                                                               <strong>Cause</strong> An attempt was made to use invalid JMS delivery parameters in the API, for example, values other than DeliveryMode.DELIVERYMODE or DeliveryMode.PERSISTENT were used to specify the delivery mode.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with delivery parameter name, such as “DeliveryMode”.{1} is replaced with delivery parameter value used by the application.</td>
<td></td>
</tr>
<tr>
<td>C4052</td>
<td><strong>Message</strong> Client ID is already in use - {0}                                                                                                                      <strong>Cause</strong> An attempt was made to set a client ID to a value that is already in use by the system.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with the client ID that is already in use.</td>
<td></td>
</tr>
<tr>
<td>C4053</td>
<td><strong>Message</strong> Invalid client ID - {0}                                                                                                                                    <strong>Cause</strong> An attempt was made to use an invalid client ID, for example, null or empty client ID.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with the client ID used by the application.</td>
<td></td>
</tr>
<tr>
<td>C4054</td>
<td><strong>Message</strong> Can not set client ID, invalid state.                                                                                                                  <strong>Cause</strong> An attempt was made to set a connection's client ID at the wrong time or when it has been administratively configured.</td>
<td></td>
</tr>
</tbody>
</table>

Warning Messages and Error Codes
<table>
<thead>
<tr>
<th>Code</th>
<th>Message and Description</th>
</tr>
</thead>
</table>
| C4055 | **Message** Resource in conflict. Concurrent operations on a session.  
|       | **Cause** An attempt was made to concurrently operate on a session with multiple threads.                                                               |
| C4056 | **Message** Received goodbye message from broker.  
|       | **Cause** A Message Queue client received a **GOOD_BYE** message from broker.                                                                          |
| C4057 | **Message** No username or password.  
|       | **Cause** An attempt was made to use a null object as a user name or password for authentication.                                                      |
| C4058 | **Message** Cannot acknowledge message for closed consumer.  
|       | **Cause** An attempt was made to acknowledge message(s) for a closed consumer.                                                                        |
| C4059 | **Message** Cannot perform operation, session is closed.  
|       | **Cause** An attempt was made to call a method on a closed session.                                                                                    |
| C4060 | **Message** Login failed: [0]  
|       | [0] message is replaced with user name.  
|       | **Cause** Login with the specified user name failed.                                                                                                    |
| C4061 | **Message** Connection recovery failed, cannot recover connection.  
|       | **Cause** The client runtime was unable to recover the connection due to internal error.                                                              |
| C4062 | **Message** Cannot perform operation, connection is closed.  
|       | **Cause** An attempt was made to call a method on a closed connection.                                                                                  |
| C4063 | **Message** Cannot perform operation, consumer is closed.  
|       | **Cause** An attempt was made to call a method on a closed message consumer.                                                                         |
| C4064 | **Message** Cannot perform operation, producer is closed.  
|       | **Cause** An attempt was made to call a method on a closed message producer.                                                                           |
| C4065 | **Message** Incompatible broker version encountered. Client version [0]. Broker version [1]  
|       | [0] is replaced with client version number. [1] is replaced with broker version number.  
|       | **Cause** An attempt was made to connect to a broker that is not compatible with the client version.                                                   |
### Table A-2  Message Queue Client Error Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>C4066</td>
<td><strong>Message</strong> Invalid or empty Durable Subscription Name was used: [0]</td>
<td>An attempt was made to use a null or empty string to specify the name of a durable subscription.</td>
</tr>
<tr>
<td></td>
<td><em>(0) is replaced with the durable subscription name used by the application.</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to use a null or empty string to specify the name of a durable subscription.</td>
<td></td>
</tr>
<tr>
<td>C4067</td>
<td><strong>Message</strong> Invalid session acknowledgment mode: [0]</td>
<td>An attempt was made to use a non-transacted session mode that is not defined in the JMS Session API.</td>
</tr>
<tr>
<td></td>
<td><em>(0) is replaced with the acknowledge mode used by the application.</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to use a non-transacted session mode that is not defined in the JMS Session API.</td>
<td></td>
</tr>
<tr>
<td>C4068</td>
<td><strong>Message</strong> Invalid Destination Classname: (0).</td>
<td>An attempt was made to create a message producer or message consumer with an invalid destination class type. The valid class type must be either <code>Queue</code> or <code>Topic</code>.</td>
</tr>
<tr>
<td></td>
<td><em>(0) is replaced with the name of the class name.</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to create a message producer or message consumer with an invalid destination class type. The valid class type must be either <code>Queue</code> or <code>Topic</code>.</td>
<td></td>
</tr>
<tr>
<td>C4069</td>
<td><strong>Message</strong> Cannot commit or rollback on an XA Session.</td>
<td>The application tried to make a <code>session.commit()</code> or a <code>session.rollback()</code> call in an application server component whose transactions are being managed by the Transaction Manager using the <code>XAResource</code>. These calls are not allowed in this context.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The application tried to make a <code>session.commit()</code> or a <code>session.rollback()</code> call in an application server component whose transactions are being managed by the Transaction Manager using the <code>XAResource</code>. These calls are not allowed in this context.</td>
<td></td>
</tr>
<tr>
<td>C4070</td>
<td><strong>Message</strong> Error when converting foreign message.</td>
<td>The client runtime encountered an error when processing a non-Message Queue JMS message.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> The client runtime encountered an error when processing a non-Message Queue JMS message.</td>
<td></td>
</tr>
<tr>
<td>C4071</td>
<td><strong>Message</strong> Invalid method in this domain: [0]</td>
<td>An attempt was made to use a method that does not belong to the current messaging domain. For example calling <code>TopicSession.createQueue()</code> will raise a <code>JMSException</code> with this error code and message.</td>
</tr>
<tr>
<td></td>
<td><em>(0) is replaced with the method name used.</em></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to use a method that does not belong to the current messaging domain. For example calling <code>TopicSession.createQueue()</code> will raise a <code>JMSException</code> with this error code and message.</td>
<td></td>
</tr>
<tr>
<td>C4072</td>
<td><strong>Message</strong> Illegal property name - &quot;&quot; or null.</td>
<td>An attempt was made to use a null or empty string to specify a property name.</td>
</tr>
<tr>
<td></td>
<td><strong>Cause</strong> An attempt was made to use a null or empty string to specify a property name.</td>
<td></td>
</tr>
<tr>
<td>C4073</td>
<td><strong>Message</strong> A JMS destination limit was reached. Too many Subscribers/Receivers for [0] : [1]</td>
<td>The client runtime was unable to create a message consumer for the specified domain and destination due to a broker resource constraint.</td>
</tr>
<tr>
<td></td>
<td>*(0) is replaced with &quot;Queue&quot; or &quot;Topic&quot; <em>(1) is replaced with the destination name.</em></td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Message</td>
<td>Cause</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>C4074</td>
<td>Transaction rolled back due to provider connection failover.</td>
<td>An attempt was made to call Session.commit() after connection failover occurred. The transaction is rolled back automatically.</td>
</tr>
<tr>
<td>C4075</td>
<td>Cannot acknowledge messages due to provider connection failover.</td>
<td>Subsequent acknowledge calls will also fail until the application calls session.recover(). As stated in the message.</td>
</tr>
<tr>
<td>C4076</td>
<td>Client does not have permission to create producer on destination: {0} {0} is replaced with the destination name that caused the exception.</td>
<td>The application client does not have permission to create a message producer with the specified destination.</td>
</tr>
<tr>
<td>C4077</td>
<td>Client is not authorized to create destination: {0} {0} is replaced with the destination name that caused the exception.</td>
<td>The application client does not have permission to create the specified destination.</td>
</tr>
<tr>
<td>C4078</td>
<td>Client is unauthorized to send to destination: {0} {0} is replaced with the destination name that caused the exception.</td>
<td>The application client does not have permission to produce messages to the specified destination.</td>
</tr>
<tr>
<td>C4079</td>
<td>Client does not have permission to register a consumer on the destination: {0} {0} is replaced with the destination name that caused the exception.</td>
<td>The application client does not have permission to create a message consumer with the specified destination name.</td>
</tr>
<tr>
<td>C4080</td>
<td>Client does not have permission to delete consumer: {0} {0} is replaced with the consumer ID for the consumer to be deleted.</td>
<td>The application does not have permission to remove the specified consumer from the broker.</td>
</tr>
<tr>
<td>C4081</td>
<td>Client does not have permission to unsubscribe: {0} {0} was replaced with the name of the subscriber to unsubscribe.</td>
<td>The client application does not have permission to unsubscribe the specified durable subscriber.</td>
</tr>
<tr>
<td>Code</td>
<td>Message</td>
<td>Cause</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>C4082</td>
<td>Client is not authorized to access destination: {0}</td>
<td>The application client is not authorized to access the specified destination.</td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with the destination name that caused the exception.</td>
<td></td>
</tr>
<tr>
<td>C4083</td>
<td>Client does not have permission to browse destination: {0}</td>
<td>The application client does not have permission to browse the specified destination.</td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with the destination name that caused the exception.</td>
<td></td>
</tr>
<tr>
<td>C4084</td>
<td>User authentication failed: {0}</td>
<td>User authentication failed.</td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with the user name.</td>
<td></td>
</tr>
<tr>
<td>C4085</td>
<td>Delete consumer failed. Consumer was not found: {0}</td>
<td>The attempt to close a message consumer failed because the broker was unable to find the specified consumer.</td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with name of the consumer that could not be found.</td>
<td></td>
</tr>
<tr>
<td>C4086</td>
<td>Unsubscribe failed. Subscriber was not found: {0}</td>
<td>An attempt was made to unsubscribe a durable subscriber with a name that does not exist in the system.</td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with name of the durable subscriber.</td>
<td></td>
</tr>
<tr>
<td>C4087</td>
<td>Set Client ID operation failed. Invalid Client ID: {0}</td>
<td>Client is unable to set Client ID on the broker and receives a BAD_REQUEST status from broker.</td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with the ClientID that caused the exception.</td>
<td></td>
</tr>
<tr>
<td>C4088</td>
<td>A JMS destination limit was reached. Too many producers for {0} : {1}</td>
<td>The client runtime was not able to create a message producer for the specified domain and destination due to limited broker resources.</td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with Queue or Topic. {1} is replaced with the destination name for which the limit was reached.</td>
<td></td>
</tr>
<tr>
<td>C4089</td>
<td>Caught JVM Error: {0}</td>
<td>The client runtime caught an error thrown from the JVM; for example, OutOfMemory error.</td>
</tr>
<tr>
<td></td>
<td>{0} is replaced with root cause error message.</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>Message</td>
<td>Cause</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| C4090  | **Message** Invalid port number. Broker is not available or may be paused:[0]  
          [0] is replaced with "[host, port]" information.  
          **Cause** The client runtime received an invalid port number (0) from the broker. Broker service for the request was not available or was paused. |
| C4091  | **Message** Cannot call Session.recover() from a NO_ACKNOWLEDGE session.  
          **Cause** The application attempts to call Session.recover() from a NO_ACKNOWLEDGE session. |
| C4092  | **Message** Broker does not support Session.NO_ACKNOWLEDGE mode, broker version: [0]  
          [0] is replaced with the version number of the broker to which the Message Queue application is connected.  
          **Cause** The application attempts to create a NO_ACKNOWLEDGE session to a broker with version # less than 3.6. |
| C4093  | **Message** Received wrong packet type. Expected: [0], but received: [1]  
          [0] is replaced with the packet type that the Message Queue client runtime expected to receive from the broker.  
          [1] is replaced with the packet type that the Message Queue client runtime actually received from the broker.  
          **Cause** The Message Queue client runtime received an unexpected Message Queue packet from broker. |
| C4094  | **Message** The destination this message was sent to could not be found: [0]  
          [0] is replaced with the destination name that caused the exception.  
          **Cause** A destination to which a message was sent could not be found. |
| C4095  | **Message** Message exceeds the single message size limit for the broker or destination: [0]  
          [0] is replaced with the destination name that caused the exception.  
          **Cause** A message exceeds the single message size limit for the broker or destination. |
| C4096  | **Message** Destination is full and is rejecting new messages: [0]  
          [0] is replaced with the destination name that caused the exception.  
          **Cause** A destination is full and is rejecting new messages. |
Index

A
acknowledge method (Message), 69
acknowledgeThisMessage method (Message), 69, 70
acknowledgeUpThroughThisMessage method (Message), 69, 70
acknowledging messages, 69, 135
acknowledgment modes, 22, 24, 41
  auto-acknowledge, 49, 105, 135
  client-acknowledge, 49, 69, 135
  defined, 48
  dups-OK-acknowledge, 49, 135
  no-acknowledge, 49, 135, 137
activation.jar file, 18
administered object store, 35
administered objects
  setProperty method, 37, 39, 46
  setProperty method, 21, 23
AdministeredObject object, 37
AdministeredObject object
  setProperty method, 37, 21, 23
AdministeredObject object
  setProperty method, 37, 39, 46
asynchronous message consumption, 25, 67
defined, 64
authentication, user, 40
AUTO_ACKNOWLEDGE constant (Session), 41, 49
auto-acknowledge mode, 105, 135
defined, 49
auto-reconnect
  behavior, 104, 106
  configurable attributes, 102
  limitations, 105

B
broker, metrics for, 149
broker acknowledgments, defined, 49
broker clusters, 30, 101, 103
  See also high availability clusters
browsing messages, 70
bytes messages
  composing, 60
  defined, 56
  getBodyLength method, 77
  processing, 76
  readBoolean method, 77
  readByte method, 77
  readBytes method, 77
  readChar method, 77
  readDouble method, 77
  readFloat method, 77
  readInt method, 77
  readLong method, 77
  readShort method, 77
  readUnsignedByte method, 77
  readUnsignedShort method, 77
  readUTF method, 77
reset method, 61
writeBoolean method, 61
writeByte method, 61
writeBytes method, 61
writeChar method, 61
writeDouble method, 61
writeFloat method, 61
writeInt method, 60
writeLong method, 61
Index

bytes messages (Continued)
  writeObject method, 61
  writeShort method, 61
  writeUTF method, 61

BytesMessage object, 56
  access methods, 77
  composition methods, 60
  getBodyLength method, 77
  readBoolean method, 77
  readByte method, 77
  readBytes method, 77
  readChar method, 77
  readDouble method, 77
  readFloat method, 77
  readInt method, 77
  readLong method, 77
  readShort method, 77
  readUnsignedByte method, 77
  readUnsignedShort method, 77
  readUTF method, 77
  reset method, 61
  writeBoolean method, 61
  writeByte method, 61
  writeBytes method, 61
  writeChar method, 61
  writeDouble method, 61
  writeFloat method, 61
  writeInt method, 60
  writeLong method, 61
  writeObject method, 61
  writeShort method, 61
  writeUTF method, 61

client applications (Continued)
  compiling, 26
  deploying, 30
  developing, 21-25
  performance, factors impacting, 94
  portability of, 79
  provider independence, 80
  running, 26
  client identifier, 66, 82
  setting, 41
  client runtime logging, 139
  client threads
    managing use of, 85
    performance, 85
  close method (Connection), 42
  close method (Connection), 23, 25, 40
  close method (MessageConsumer), 65, 71
  close method (MessageProducer), 62, 64
  close method (QueueBrowser), 70, 71
  close method (Session), 23, 25
  close method (Session), 48
  clustered broker configuration, 30, 101, 103
  com.sun.messaging.jms package, 49, 70
  com.sun.messaging package, 38, 40
  command line, -D option, 37
  commit method (Session), 48, 51
  committing transactions, 51
    defined, 50
configuration properties
  connection factory, overriding, 37, 39
  imqAddressList, 101, 102
  imqAddressListBehavior, 101, 102, 104
  imqAddressListIterations, 102
  imqDefaultPassword, 37
  imqDestinationDescription, 46
  imqDestinationName, 46
  imqPingInterval, 93
  imqReconnectAttempts, 102
  imqReconnectEnabled, 101
  imqReconnectInterval, 102
connection events, 98
connection factories
  createConnection method, 21, 24
  createConnection method, 39

C
C clients, communicating with, 139
CLASSPATH environment variable, 17, 19, 29
clearBody method (Message), 57, 58, 61
CLIENT_ACKNOWLEDGE constant (Session), 41, 49
client-acknowledge mode, 69, 135
  defined, 49
client acknowledgments, defined, 49
client applications
  avoiding deadlock, 85
connection factories (Continued)
createQueueConnection method, 40
createTopicConnection method, 40
defined, 34
_imqAddressList configuration property, 39
instantiating, 21, 23, 38
JNDI lookup, 21, 23, 35
overriding configuration properties, 37, 39
read-only, 38
Connection object, 21, 24, 34
Connection object
close method, 42
Connection object
close method, 23, 25, 40
createConnectionConsumer method, 40, 41
createDurableConnectionConsumer method, 40, 41
Connection object
createSession method, 40
Connection object
createSession method, 22, 24, 40, 48, 49, 50
creating, 21, 24, 39
default user identity, 40
defined, 34
event notifications, 98
exception listener, 41
getExceptionListener method, 40
getMetaData method, 40, 41
methods, 40
setClientID method, 40, 41
setEventListener method, 40
setExceptionListener method, 40, 41
Connection object
start method, 67
Connection object
start method, 24, 40, 41, 42, 68
stop method, 40, 41
ConnectionClosedEvent, 99
ConnectionClosingEvent, 99
ConnectionConfiguration object, 37
ConnectionFactory object, 18, 21, 23, 34, 38, 40
ConnectionFactory object
createConnection method, 21, 24
ConnectionFactory object
createConnection method, 39
createQueueConnection method, 40
createTopicConnection method, 40
ConnectionMetaData object, 41
ConnectionReconnectedEvent, 99
ConnectionReconnectFailedEvent, 99
collections
authentication, 40
client identifier, 41, 66
close method, 42
close method, 23, 25, 40
closing, 23, 25, 42
createConnectionConsumer method, 40, 41
createDurableConnectionConsumer method, 40, 41
createSession method, 40
createSession method, 22, 24, 40, 48, 49, 50
creating, 21, 24, 39
default user identity, 40
defined, 34
event notifications, 98
exception listener, 41
getExceptionListener method, 40
getMetaData method, 40, 41
password, 40
reconnecting, 101
secure, 42
setClientID method, 40, 41
setEventListener method, 40
setExceptionListener method, 40, 41
setting default password, 37
start method, 24, 40, 41, 42, 68
start method, 67
starting, 24, 41, 67
stop method, 40, 41
thread use by, 85
constants
AUTO_ACKNOWLEDGE (Session), 41, 49
CLIENT_ACKNOWLEDGE (Session), 41, 49
DEFAULT_DELIVERY_MODE (Message), 53
DEFAULT_PRIORITY (Message), 54
DEFAULT_TIME_TO_LIVE (Message), 54
DUPS_OK_ACKNOWLEDGE (Session), 41, 49
NO_ACKNOWLEDGE (Session), 41, 49
NON_PERSISTENT (DeliveryMode), 53
PERSISTENT (DeliveryMode), 53
SESSION_TRANSACTED (Session), 50, 69
correlation identifier, defined, 53
createBrowser method (Session), 48, 70
createBytesMessage method (Session), 48, 56
createBytesMessage method (Session), 60
createConnection method (ConnectionFactory), 21, 24
createConnection method (ConnectionFactory), 39
createConnectionConsumer method (Connection), 40
createDurableMessageConsumer method (Connection), 40
createDurableSubscriber method (Session), 47, 66, 67, 68
createDurableConnectionConsumer method (Connection), 40, 41
createDurableSubscriber method (Session), 47, 66, 67, 68
createMapMessage method (Session), 47, 56
createMessage method (Session), 47, 56
createMessage method (Session), 47, 56, 60
createProducer method (Session), 22
createPublisher method (TopicSession), 63, 64
createQueue method (Session), 48
createQueueConnection method (ConnectionFactory), 40
derelation, 64
delete method (TemporaryQueue, TemporaryTopic), 47
delivery modes, 50, 95
default, message producer, 63
defined, 53
nonpersistent, 54
persistent, 53
setting, 63
DeliveryMode object, 53
NON_PERSISTENT constant, 53
PERSISTENT constant, 53
deploying client applications, 30
destination metrics, 151
Destination object, 18, 22, 24, 34, 43, 53
DestinationConfiguration object, 46
destinations
default, message producer, 62, 63
defined, 43
instantiating, 22, 24, 46
JNDI lookup, 22, 24, 43
message, setting, 63
queue, 22, 24, 33, 43, 62, 63, 64, 70
temporary, 47
topic, 22, 24, 34, 43, 62, 63, 64, 66
distributed applications and synchronous consumers, 93
dups-OK_acknowledge mode, 135
defined, 49
durable subscribers, 64
client identifier, 66
closing, 71
defined, 66
subscriber name, 66
durable subscriptions, 41
defined, 33
performance impact of, 96-97
dead message queue, 88
DEFAULT_DELIVERY_MODE constant (Message), 53
DEFAULT_PRIORITY constant (Message), 54
DEFAULT_TIME_TO_LIVE constant (Message), 54
default user identity, 40
delete method (TemporaryQueue, TemporaryTopic), 47
delivery modes, 50, 95
default, message producer, 63
defined, 53
nonpersistent, 54
persistent, 53
setting, 63
DeliveryMode object, 53
NON_PERSISTENT constant, 53
PERSISTENT constant, 53
deploying client applications, 30
destination metrics, 151
Destination object, 18, 22, 24, 34, 43, 53
DestinationConfiguration object, 46
destinations
default, message producer, 62, 63
defined, 43
instantiating, 22, 24, 46
JNDI lookup, 22, 24, 43
message, setting, 63
queue, 22, 24, 33, 43, 62, 63, 64, 70
temporary, 47
topic, 22, 24, 34, 43, 62, 63, 64, 66
distributed applications and synchronous consumers, 93
dups-OK_acknowledge mode, 135
defined, 49
durable subscribers, 64
client identifier, 66
closing, 71
defined, 66
subscriber name, 66
durable subscriptions, 41
defined, 33
performance impact of, 96-97
dead message queue, 88
DEFAULT_DELIVERY_MODE constant (Message), 53
DEFAULT_PRIORITY constant (Message), 54
DEFAULT_TIME_TO_LIVE constant (Message), 54
default user identity, 40
delete method (TemporaryQueue, TemporaryTopic), 47
delivery modes, 50, 95
default, message producer, 63
defined, 53
nonpersistent, 54
persistent, 53
setting, 63
DeliveryMode object, 53
NON_PERSISTENT constant, 53
PERSISTENT constant, 53
deploying client applications, 30
destination metrics, 151
Destination object, 18, 22, 24, 34, 43, 53
DestinationConfiguration object, 46
destinations
default, message producer, 62, 63
defined, 43
instantiating, 22, 24, 46
JNDI lookup, 22, 24, 43
message, setting, 63
queue, 22, 24, 33, 43, 62, 63, 64, 70
temporary, 47
topic, 22, 24, 34, 43, 62, 63, 64, 66
distributed applications and synchronous consumers, 93
event listener, 98, 99
event notification API, 98
example programs, 31
HelloWorldMessage, 26
exception listeners, 41
onException method, 41
ExceptionListener object, 41
onException method, 41
exceptions
InvalidDestinationException, 70
MessageFormatException, 58, 59, 60
MessageNotWriteableException, 57, 58, 61
expiration time (message), 54

file-system object store, 36, 44
FLOW_CONTROL property, 92
fscontext.jar file, 19

getAcknowledgeMode method (Session), 48, 49, 50, 69
getBodyLength method (BytesMessage), 77
getBoolean method (MapMessage), 75
getBooleanProperty method (Message), 72
getByte method (MapMessage), 75
getByteProperty method (Message), 72
getBytes method (MapMessage), 75
getChar method (MapMessage), 75
getClientID method (Connection), 40, 41
getDeliveryMode method (MessageProducer), 62, 63
getDestination method (MessageProducer), 62
getDisableMessageID method (MessageProducer), 62
getDisableMessageTimestamp method (MessageProducer), 62
getDouble method (MapMessage), 75
getDoubleProperty method (Message), 72
getFloat method (MapMessage), 75
getFloatProperty method (Message), 72
getInt method (MapMessage), 75
getIntProperty method (Message), 72
getJMSCorrelationIDAsBytes method (Message), 72
getJMSDeliveryMode method (Message), 72
getJMSDestination method (Message), 72
getJMSExpiration method (Message), 72
getJMSMessageID method (Message), 71
getJMSPriority method (Message), 25
getJMSRedelivered method (Message), 72
getJMSReplyTo method (Message), 72
getJMSTimestamp method (Message), 72
getJMSType method (Message), 72
getLong method (MapMessage), 75
getLongProperty method (Message), 72
getMapNames method (MapMessage), 75
getMapNames method (MapMessage), 76
getMessageListener method (MessageConsumer), 65, 69
getMessageListener method (Session), 41, 48
getMessageSelector method (MessageConsumer), 65
getMessageSelector method (QueueBrowser), 70
getMetaData method (Connection), 40, 41
getObject method (MapMessage), 75, 76
goal method (ObjectMessage), 76
getObjectProperty method (Message), 72, 73
genPriority method (MessageProducer), 62, 63
getPropertyNames method (Message), 73
goalQueue method (QueueBrowser), 70
goalShort method (MapMessage), 75
goalShortProperty method (Message), 72
goalString method (MapMessage), 75
goalStringProperty method (Message), 25
goalStringProperty method (Message), 72
goalText method (TextMessage), 25
goalText method (TextMessage), 73, 74
goalTimeToLive method (MessageProducer), 62, 63
goalTransacted method (Session), 48, 50

hash table for destination-list metrics, 157
heap space, JVM, 86
HelloWorldMessage example program, 26
high availability clusters
  address list, effect on, 102
effect on design, 101, 104
reconnection attempts, 102
reconnection behavior, 104

I
imq.jar file, 18
imqAddressList configuration property, 39, 101, 102
imqAddressListBehavior configuration property, 101, 102, 104
imqAddressListIterations configuration property, 102
imqbrokerd command, 19
imqcmd command, 20
imqDefaultPassword configuration property, 37
imqDestinationDescription configuration property, 46
imqDestinationName configuration property, 46
imqPingInterval configuration property, 93
imqReconnectAttempts configuration property, 102
imqReconnectEnabled configuration property, 101
imqReconnectInterval configuration property, 102
imqm.jar file, 18
InvalidDestinationException exception, 70
itemExists method (MapMessage), 75

J
JAF, See JavaBeans Activation Framework
.jar files, 17
  activation.jar, 18
  for JMS and SOAP clients, 18
  fscontext.jar, 19
  imq.jar, 18
  imq.jar, 18
  imqm.jar, 18
  jaxm-api.jar, 18
  jms.jar, 18
  jndi.jar, 18, 19
  ldap.jar, 19
  locations, 18
  needed in CLASSPATH, 18
  saaj-api.jar, 18
Java Development Kit (JDK), 17, 18, 19
Java Message Service Specification, 7, 33, 41, 52, 54, 65
Java Naming and Directory Interface (JNDI), 35
  environment parameter, 36, 45
  initial context, 37, 45
  initial context factory, 36, 45
  .jar file needed for, 18
  obtaining connection factories with, 21, 23, 35
  obtaining destinations with, 22, 24, 43
java.util.logging library, 139
java.util package, 29
Java Virtual Machine
  heap space, 86
  metrics for, 150
JavaBeans Activation Framework (JAF), 176
javax.jms package, 29, 38, 40, 46, 49
javax.xml.messaging package, 167
javax.xml.soap package, 167
jaxm-api.jar file, 18
JAXMServlet object, 179
JDK
  See Java Development Kit
jms.jar file, 18
JMS_SUN_COMPRESS property, 87
JMS_SUN_COMPRESSED_SIZE property, 87
JMS_SUN_DMQ_BODY_TRUNCATED property, 92
JMS_SUN_DMQ_DEAD_BROKER property, 91
JMS_SUN_DMQ_PRODUCING_BROKER property, 91
JMS_SUN_DMQ_UNDELIVERED_COMMENTS property, 91
JMS_SUN_DMQ_UNDELIVERED_EXCEPTION property, 91
JMS_SUN_DMQ_UNDELIVERED_REASON property, 91
JMS_SUN_DMQ_UNDELIVERED_TIMESTAMP property, 91
JMS_SUN_LOG_DEAD_MESSAGES property, 90
JMS_SUN_PRESERVE_UNDELIVERED property, 89
JMS_SUN_TRUNCATE_MSG_BODY property, 90
JMS_SUN_UNCOMPRESSED_SIZE property, 87
JMSCorrelationID message header field, 52, 53
JMSDeliveryMode message header field, 52, 53
JMSDestination message header field, 52, 53
JMSExpiration message header field, 52, 54
JMSMessageID message header field, 52, 53
JMSPriority message header field, 52, 54
JMSRedelivered message header field, 52, 54
JMSReplyTo message header field, 52, 53, 62
JMSTimestamp message header field, 52, 54
JMSType message header field, 52, 54
JMSAppID message property, 55
JMSConsumerTXID message property, 55
JMSDeliveryCount message property, 55
JMSDeliveryCount property, 91
JMSGroupID message property, 55
JMSGroupSeq message property, 55
JMSProducerTXID message property, 55
JMSRecoverTimestamp message property, 55
JMSState message property, 55
JMSUserID message property, 55
JNDI, See Java Naming and Directory Interface
jndi.jar file, 18, 19
JRE_DIRECTORY/lib/logging.properties file, 142
JVM, See Java Virtual Machine

M
map messages
  composing, 59
  defined, 56
  getBoolean method, 75
  getByte method, 75
  getBytes method, 75
  getChar method, 75
  getDouble method, 75
  getFloat method, 75
  getInt method, 75
  getLong method, 75
  getMapNames method, 75, 76
  getobject method, 75, 76
  getShort method, 75
  getString method, 75
  itemExists method, 75
  processing, 75
  setBoolean method, 59
  setByte method, 59
  setBytes method, 59
  setChar method, 59
  setDouble method, 59
  setFloat method, 59
  setInt method, 59
  setLong method, 59
  setObject method, 59
  setShort method, 59
  setString method, 59
MapMessage object, 56
  access methods, 75
  composition methods, 59
  getBoolean method, 75
  getByte method, 75
  getBytes method, 75
  getChar method, 75
  getDouble method, 75
  getFloat method, 75
  getInt method, 75
  getLong method, 75
  getMapNames method, 75, 76
  getObject method, 75, 76
  getShort method, 75
  getString method, 75

L
ldabbp.jar file, 19
LDAP, See Lightweight Directory Access Protocol
ldap.jar file, 19
lib directory, 17
lifetime (message), 54
  default, message producer, 63
  setting, 63
Lightweight Directory Access Protocol (LDAP)
  .jar files needed for, 19
  object store, 36, 44
logging
  client runtime, for, 139
  configuration file, 143
  events, 141
  JRE_DIRECTORY/lib/logging.properties file, 142
  levels, 140
  name space for Message Queue, 140
<table>
<thead>
<tr>
<th>MapMessage object (Continued)</th>
<th>message header (Continued)</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemExists method, 75</td>
<td>JMSPriority field, 52, 54</td>
</tr>
<tr>
<td>setBoolean method, 59</td>
<td>JMSRedelivered field, 52, 54</td>
</tr>
<tr>
<td>setByte method, 59</td>
<td>JMSReplyTo field, 52, 53, 62</td>
</tr>
<tr>
<td>setBytes method, 59</td>
<td>JMSTimestamp field, 52, 54</td>
</tr>
<tr>
<td>setChar method, 59</td>
<td>JMSType field, 52, 54</td>
</tr>
<tr>
<td>setDouble method, 59</td>
<td>retrieving fields, 71</td>
</tr>
<tr>
<td>setFloat method, 59</td>
<td>message identifier</td>
</tr>
<tr>
<td>setInt method, 59</td>
<td>defined, 53</td>
</tr>
<tr>
<td>setLong method, 59</td>
<td>suppressing, 63</td>
</tr>
<tr>
<td>setObject method, 59</td>
<td>message listeners, 25, 64</td>
</tr>
<tr>
<td>setShort method, 59</td>
<td>creating, 67</td>
</tr>
<tr>
<td>setString method, 59</td>
<td>onMessage method, 68, 69</td>
</tr>
<tr>
<td>master broker, 104</td>
<td>Message object, 56</td>
</tr>
<tr>
<td>memory management, 86</td>
<td>acknowledge method, 69</td>
</tr>
<tr>
<td>message-based monitoring, 145</td>
<td>acknowledgeThisMessage method, 69, 70</td>
</tr>
<tr>
<td>message body</td>
<td>acknowledgeUpThroughThisMessage method, 69, 70</td>
</tr>
<tr>
<td>defined, 56</td>
<td>acknowledgment methods, 69</td>
</tr>
<tr>
<td>processing, 73</td>
<td>clearBody method, 57, 58, 61</td>
</tr>
<tr>
<td>message brokers</td>
<td>clearProperties method, 54, 55</td>
</tr>
<tr>
<td>starting, 19</td>
<td>DEFAULT_DELIVERY_MODE constant, 53</td>
</tr>
<tr>
<td>testing, 20-21</td>
<td>DEFAULT_PRIORITY constant, 54</td>
</tr>
<tr>
<td>message consumers</td>
<td>DEFAULT_TIME_TO_LIVE constant, 54</td>
</tr>
<tr>
<td>close method, 65, 71</td>
<td>getBooleanProperty method, 72</td>
</tr>
<tr>
<td>closing, 71</td>
<td>getByteProperty method, 72</td>
</tr>
<tr>
<td>creating, 24, 64</td>
<td>getDoubleProperty method, 72</td>
</tr>
<tr>
<td>dedicated, 92</td>
<td>getFloatProperty method, 72</td>
</tr>
<tr>
<td>defined, 64</td>
<td>getIntProperty method, 72</td>
</tr>
<tr>
<td>getMessageListener method, 65, 69</td>
<td>getJMSCorrelationID method, 72</td>
</tr>
<tr>
<td>getMessageSelector method, 65</td>
<td>getJMSCorrelationIDAsBytes method, 72</td>
</tr>
<tr>
<td>message loss, correcting, 93</td>
<td>getJMSDeliveryMode method, 72</td>
</tr>
<tr>
<td>pinging, 92</td>
<td>getJMSDestination method, 72</td>
</tr>
<tr>
<td>receive method, 25, 65, 67, 69</td>
<td>getJMSExpiration method, 72</td>
</tr>
<tr>
<td>receiveNoWait method, 65, 67, 69</td>
<td>getJMSMessageID method, 71</td>
</tr>
<tr>
<td>setMessageListener method, 65, 68</td>
<td>getJMSPriority method, 72</td>
</tr>
<tr>
<td>synchronous, 93</td>
<td>Message object</td>
</tr>
<tr>
<td>message delivery models, 80</td>
<td>getJMSRedelivered method, 72</td>
</tr>
<tr>
<td>message header</td>
<td>getJMSReplyTo method, 72</td>
</tr>
<tr>
<td>defined, 52</td>
<td>getJMSTimestamp method, 72</td>
</tr>
<tr>
<td>JMSCorrelationID field, 52, 53</td>
<td>getJMSType method, 72</td>
</tr>
<tr>
<td>JMSDeliveryMode field, 52, 53</td>
<td>getLongProperty method, 72</td>
</tr>
<tr>
<td>JMSDestination field, 52, 53</td>
<td>Message object</td>
</tr>
<tr>
<td>JMSExpiration field, 52, 54</td>
<td>getJMSRedelivered method, 72</td>
</tr>
<tr>
<td>JMSSpendingID field, 52, 53</td>
<td>getJMSReplyTo method, 72</td>
</tr>
<tr>
<td>JMSTimestamp field, 52, 54</td>
<td>getJMSTimestamp method, 72</td>
</tr>
<tr>
<td>JMSType field, 52, 54</td>
<td>getJMSType method, 72</td>
</tr>
<tr>
<td>JMSType field, 52, 54</td>
<td>getLongProperty method, 72</td>
</tr>
</tbody>
</table>
Message object (Continued)
   getBooleanProperty method, 73
   getByteProperty method, 55
   getDoubleProperty method, 55
   getFloatProperty method, 55
   getIntProperty method, 55
   setBooleanProperty method, 55
   setByteProperty method, 55
   setDoubleProperty method, 55
   setFloatProperty method, 55
   setIntProperty method, 55
   setJMSCorrelationID method, 53
   setJMSCorrelationIDAsBytes method, 53
   setJMSDeliveryMode method, 62,63
   setJMSDestination method, 62
   setJMSRedelivered method, 53
   setJMSReplyTo method, 53
   setJMSExpiration method, 53
   setJMSMessageID method, 52
   setJMSMessageIDAsBytes method, 53
   setJMSExpiration method, 62,63
   setJMSDeliveryMode method, 62,63
   setMessageIDAsBytes method, 53
   setPriority method, 62,63
   getTimeToLive method, 62,63
   send method, 23,62,63
   setPriority method, 62,63
   setJMSRedelivered method, 62,63
   setJMSDeliveryMode method, 62,63
   setJMSMessageID method, 53
   setJMSMessageIDAsBytes method, 53
   setJMSExpiration method, 54,62,63
   setJMSDeliveryMode method, 62,63
   message producers (Continued)
   getDestination method, 62
   getDisableMessageID method, 62
   getDisableMessageTimestamp method, 62
   getPriority method, 62,63
   getTimeToLive method, 62,63
   send method, 23,62,63
   setDeliveryMode method, 62,63
   setDisableMessageID method, 53,62,63
   setDisableMessageTimestamp method, 54
   setJMSDeliveryMode method, 62,63
   setJMSMainEntity method, 62,63
   message properties
      defined, 54
      filtering on, 65
      retrieving, 72
   message selectors
      browsing with, 70
      defined, 65
      efficient use of, 83
      performance impact, 97-98
   message type identifier, defined, 54
   message types, defined, 56
   MessageConsumer object, 24,34,64,66
      close method, 65,71
      setMessageListener method, 65,69
      setMessageSelector method, 65
      methods, 65
      receive method, 25,65,67,69
      receiveNoWait method, 65,67,69
      setMessageListener method, 65,68
   MessageFactory objectd, 181
   MessageFormatException exception, 58,59,60
   MessageListener object, 67,68
      onMessage method, 68,69
   MessageNotWriteableException exception, 57,58,61
   MessageProducer object, 22,34,63
      close method, 62,64
      getDeliveryMode method, 62,63
      getDestination method, 62
      getDisableMessageID method, 62
      getDisableMessageTimestamp method, 62
      getPriority method, 62,63
      getTimeToLive method, 62,63
      getDeliveryMode method, 62,63
      getDestination method, 62
      getDisableMessageID method, 62
      getDisableMessageTimestamp method, 62
      getPriority method, 62,63
      getTimeToLive method, 62,63
MessageProducer object (Continued)
methods, 62
send method, 23, 62, 63
setDeliveryMode method, 62, 63
setDisableMessageID method, 53, 62, 63
setDisableMessageTimestamp method, 54, 62, 63
setPriority method, 62, 63
setTimeToLive method, 62, 63
messages
acknowledge method, 69
acknowledgeThisMessage method, 69, 70
acknowledgingUpThroughThisMessage method, 69, 70
acknowledging, 69, 135
body, 56, 73
body type and performance, 98
browsing, 70
bytes messages, 56, 60, 76
clearBody method, 57, 58, 61
clearProperties method, 54, 55
composing, 56
compression, 82, 87
correlation identifier, 53
creating, 22, 56
delivery mode, 63
delivery models, 80
destination, 63
encrypting, 42
expiration time, 54
filtering, 65
getBooleanProperty method, 72
getByteProperty method, 72
getDoubleProperty method, 72
getFloatProperty method, 72
getIntProperty method, 72
getJMSCorrelationID method, 72
getJMSCorrelationIDAsBytes method, 72
getJMSDeliveryMode method, 72
getJMSDestination method, 72
getJMSExpiration method, 72
getJMSMessageID method, 71
getJMSPriority method, 25
getJMSPriority method, 72
getJMSRedelivered method, 72
messages (Continued)
getJMSReplyTo method, 72
getJMSTimestamp method, 72
getJMSType method, 72
getLongProperty method, 72
getObjectProperty method, 72, 73
getPropertyNames method, 73
getShortProperty method, 72
getStringProperty method, 25
getStringProperty method, 72
header, 52, 71
JMSXAppID property, 55
JMSXConsumerTXID property, 55
JMSXDeliveryCount property, 55
JMSXGroupID property, 55
JMSXGroupSeq property, 55
JMSXProducerTXID property, 55
JMSXRcvTimestamp property, 55
JMSXState property, 55
JMSXUserID property, 55
lifetime, 54, 63
map messages, 56, 59, 75
message identifier, 53, 63
message type identifier, 54
message types, 56
null messages, 56
object messages, 56, 60, 76
ordering of, 83
prioritizing, 83
priority, 54, 63
processing, 71
properties, 54, 72
propertyExists method, 73
receiving, 25, 64
redelivered flag, 54
retrieving content, 25
retrieving header fields, 25
retrieving properties, 25
sending, 23, 61
sequencing, 86
setBooleanProperty method, 55
setByteProperty method, 55
setDoubleProperty method, 55
setFloatProperty method, 55
messages (Continued)
setIntProperty method, 55
setJMSCorrelationID method, 53
setJMSCorrelationIDAsBytes method, 53
setJMSDeliveryMode method, 53
setJMSDestination method, 52
setJMSExpiration method, 53
setJMSMessageID method, 52
setJMSPriority method, 53
setJMSRedelivered method, 53
setJMSReplyTo method, 53
setJMSTimestamp method, 53
setJMSReplyTo method, 53
methods (Continued)
setLongProperty method, 55
setObjectProperty method, 54, 55
setShortProperty method, 55
setStringProperty method, 23
setting content, 22
setting properties, 22
size, and performance, 97
size of, 86, 87
SOAP payloads, with, 185
standard properties, 55
stream messages, 56, 57, 74
structure, 51
text messages, 56, 57, 73
time stamp, 54, 63
messaging domains, 33, 80-81
point-to-point (PTP), 22, 24, 33, 40
publish/subscribe (pub/sub), 22, 24, 34, 40, 41
unified, 34, 40, 43
methods
acknowledge (Message), 69
acknowledgeThisMessage (Message), 69, 70
acknowledgeUpThroughThisMessage (Message), 69, 70
clearBody (Message), 57, 58, 61
close (Connection), 42
close (Connection), 23, 25, 40
close (MessageConsumer), 65, 71
close (MessageProducer), 62, 64
close (QueueBrowser), 70, 71
close (Session), 23, 25
methods (Continued)
getDestination (MessageProducer), 62
getDisableMessageID (MessageProducer), 62
getDisableMessageTimestamp (MessageProducer), 62
getDouble (MapMessage), 75
getDoubleProperty (Message), 72
genEnumeration (QueueBrowser), 72
genExceptionListener (Connection), 40
genFloat (MapMessage), 75
genFloatProperty (Message), 72
getInt (MapMessage), 75
getIntProperty (Message), 72
genJMSCorrelationID (Message), 72
genJMSCorrelationIDAsBytes (Message), 72
genJMSDeliveryMode (Message), 72
genJMSDestination (Message), 72
genJMSExpiration (Message), 72
genJMSMessageID (Message), 71
genJMSPriority (Message), 25
genJMSRedelivered (Message), 72
genJMSReplyTo (Message), 72
genJMSTimestamp (Message), 72
genJMSType (Message), 72
getLong (MapMessage), 75
genLongProperty (Message), 72
getMapNames (MapMessage), 75, 76
getMessageListener (MessageConsumer), 65, 69
getMessageListener (Session), 41, 48
getMessageSelector (MessageConsumer), 65
getMessageSelector (QueueBrowser), 70
gEtMetaData (Connection), 40, 41
getObject (MapMessage), 75, 76
genObject (ObjectMessage), 76
genObjectProperty (Message), 72, 73
genPriority (MessageProducer), 62, 63
getPropertyNames (Message), 73
genQueue (QueueBrowser), 70
genQueue (Message), 75
genQueueProperty (Message), 25
genStringProperty (Message), 72
methods (Continued)
getText (TextMessage), 25
genText (TextMessage), 73, 74
genTimeToLive (MessageProducer), 62, 63
genTransacted (Session), 48, 50
itemExists (MapMessage), 75
onException (ExceptionListener), 41
onMessage (MessageListener), 68, 69, 181
propertyExists (Message), 73
Queue constructor, 46
readBoolean (BytesMessage), 77
readBoolean (StreamMessage), 74
readByte (BytesMessage), 77
readByte (StreamMessage), 74
readBytes (BytesMessage), 77
readBytes (StreamMessage), 74
readChar (BytesMessage), 77
readChar (StreamMessage), 74
readDouble (BytesMessage), 77
readDouble (StreamMessage), 74
readFloat (BytesMessage), 77
readFloat (StreamMessage), 74
readInt (BytesMessage), 77
readInt (StreamMessage), 74
readLong (BytesMessage), 77
readLong (StreamMessage), 74
readObject (StreamMessage), 77
readShort (BytesMessage), 77
readShort (StreamMessage), 74
readString (StreamMessage), 74
readUnsignedByte (BytesMessage), 77
readUnsignedShort (BytesMessage), 77
readUTF (BytesMessage), 77
receive (MessageConsumer), 25, 65, 67, 69
receiveNoWait (MessageConsumer), 65, 67, 69
recover (Session), 48
reset (BytesMessage), 61
reset (StreamMessage), 58
rollback (Session), 48, 51
send (MessageProducer), 23, 62, 63
setBoolean (MapMessage), 59
setBooleanProperty (Message), 55, 87
setByte (MapMessage), 59
setByteProperty (Message), 55
setCharProperty (Message), 55
setDoubleProperty (Message), 55
setFloatProperty (Message), 55
setIntProperty (Message), 55
setLongProperty (Message), 55
setShortProperty (Message), 55
setStringProperty (Message), 55
setTimeToLive (Session), 48, 50
setTransacted (Session), 48, 50
methods (Continued)
setBytes (MapMessage), 59
setChar (MapMessage), 59
setClientID (Connection), 40, 41
setDeliveryMode (MessageProducer), 62, 63
setDisableMessageID (MessageProducer), 53, 62, 63
setDisableMessageTimestamp (MessageProducer), 54, 62, 63
setDoubleProperty (Message), 55
setEventListener (Connection), 40
setExceptionListener (Connection), 40, 41
setFloat (MapMessage), 59
setFloatProperty (Message), 55
setInt (MapMessage), 59
setIntProperty (Message), 55
setJMSCorrelationID (Message), 53
setJMSCorrelationIDAsBytes (Message), 53
setJMSDeliveryMode (Message), 53
setJMSDestination (Message), 52
setJMSExpiration (Message), 53
setJMSMessageID (Message), 52, 54, 55
setJMSPriority (Message), 53
setJMSRedelivered (Message), 53
setJMSReplyTo (Message), 53
setJMSTimestamp (Message), 53
setJMSType (Message), 53
setLong (MapMessage), 59
setLongProperty (Message), 55
setMessageListener (MessageConsumer), 65, 68
setMessageListener (Session), 41, 48
setObject (MapMessage), 59
setObject (ObjectMessage), 60
setObjectProperty (Message), 54, 55
setPriority (MessageProducer), 62, 63
setProperty (AdministeredObject), 21, 23
setShort (MapMessage), 59
setShortProperty (Message), 55
setString (MapMessage), 59
setStringProperty (Message), 55
setStringProperty (Message), 23
setText (TextMessage), 57
setText (TextMessage), 22
methods (Continued)
setTimeToLive (MessageProducer), 62, 63
start (Connection), 24, 40, 41, 42, 68
start (Connection), 67
stop (Connection), 40, 41
Topic constructor, 46
unsubscribe (Session), 47, 67, 71
writeBoolean (BytesMessage), 61
writeBoolean (StreamMessage), 58
writeByte (BytesMessage), 61
writeByte (StreamMessage), 58
writeBytes (BytesMessage), 61
writeBytes (StreamMessage), 58
writeChar (BytesMessage), 61
writeChar (StreamMessage), 58
writeDouble (BytesMessage), 61
writeDouble (StreamMessage), 58
writeFloat (BytesMessage), 61
writeFloat (StreamMessage), 58
writeInt (BytesMessage), 60
writeInt (MapMessage), 59
writeInt (StreamMessage), 58
writeLong (BytesMessage), 61
writeLong (StreamMessage), 58
writeObject (BytesMessage), 61
writeObject (StreamMessage), 58
writeShort (BytesMessage), 61
writeShort (StreamMessage), 58
writeString (StreamMessage), 58
writeUTF (BytesMessage), 61
metrics-based monitoring
administration of, 147
creating client for, 148
examples of, 153
implementation of, 147
introduced, 145
metrics messages
format of, 146, 149
properties of, 149
type, 146
MimeHeaders object, 181
mq.metrics.broker topic, 146
mq.metrics.destination_list topic, 146
mq.metrics.destination.queue.dn topic, 146
objects (Continued)
MapMessage, 56, 59, 75
Message, 52, 54, 56, 69, 71, 72
MessageConsumer, 24, 34, 64, 65, 66
MessageFactory, 181
MessageListener, 67, 68
MessageProducer, 22, 34, 62, 63
MimeHeaders, 181
ObjectNameMessage, 56, 60, 76
Queue, 29, 34, 43
QueueBrowser, 70
QueueConnection, 34
QueueConnectionFactory, 34
QueueReceiver, 34, 64
QueueSender, 34, 63
QueueSession, 34, 63, 64
ReqRespListener, 175
ServletConfig, 180
Session, 22, 24, 34, 47, 56
StreamMessage, 56, 57, 74
TextMessage, 56, 73
Topic, 34, 43
TopicConnection, 34
TopicConnectionFactory, 34
TopicPublisher, 34, 63
TopicSession, 34, 63, 64
TopicSubscriber, 34, 64, 66
URLEndpoint, 179
onException method (ExceptionListener), 41
onMessage method (MessageListener), 68, 69, 181
OutOfMemoryError error, 86

P
packages
com.sun.messaging, 38, 40
com.sun.messaging.jms, 49, 70
java.util, 29
javax.jms, 29, 38, 40, 46, 49
javax.xml.messaging, 167
javax.xml.soap, 167
passwords
connection, authentication, 40
connection, setting, 37
performance, factors impacting
acknowledgment mode, 96
delivery mode, 95
durable subscriptions, 96-97
message body type, 98
message selectors, 97-98
message size, 97
transactions, 95
performance and reliability, 94
PERSISTENT constant (DeliveryMode), 53
persistent delivery mode, defined, 53
physical destination properties, 92
ping interval, 92
point-to-point (PTP) messaging domain, 22, 24, 40
  defined, 33
priority (message)
  default, message producer, 63
  defined, 54
  setting, 63
processing messages, 71
propertyExists method (Message), 73
provider independence, 80
PTP, See point-to-point messaging domain
pub/sub, See publish/subscribe messaging domain
publish/subscribe (pub/sub) messaging domain, 22, 24, 40, 41
  defined, 33
queue sessions
  createReceiver method, 64
  createSender method, 63
QueueBrowser object
  close method, 70, 71
  getMessageSelector method, 70
  getQueue method, 70
  methods, 70
QueueConnection object, 34
QueueConnectionFactory object, 34
QueueReceiver object, 34, 64
QueueSender object, 34, 63
QueueSession object, 34, 63, 64
  createReceiver method, 64
  createSender method, 63
R
readBoolean method (BytesMessage), 77
readBoolean method (StreamMessage), 74
readByte method (BytesMessage), 77
readByte method (StreamMessage), 74
readBytes method (BytesMessage), 77
readBytes method (StreamMessage), 74
readChar method (BytesMessage), 77
readChar method (StreamMessage), 74
readDouble method (BytesMessage), 77
readDouble method (StreamMessage), 74
readFloat method (BytesMessage), 77
readFloat method (StreamMessage), 74
readInt method (BytesMessage), 77
readInt method (StreamMessage), 74
readLong method (BytesMessage), 77
readLong method (StreamMessage), 74
readObject method (StreamMessage), 74
readObject method (StreamMessage), 74
readObject method (StreamMessage), 74
readString method (StreamMessage), 74
readUnsignedByte method (BytesMessage), 77
readUnsignedShort method (BytesMessage), 77
readUTF method (BytesMessage), 77
receive method (MessageConsumer), 25, 65, 67, 69
receiveNoWait method (MessageConsumer), 65, 67, 69
receiving messages, 25, 64
Index

receiving messages (Continued)
  asynchronously, 67
  synchronously, 67
recover method (Session), 48
redelivered flag (message), defined, 54
REJECT_NEWEST property, 92
reliability and performance, 94
REMOVE_LOW_PRIORITY property, 92
REMOVE_OLEDEST property, 92
ReqRespListener object, 175
reset method (BytesMessage), 61
reset method (StreamMessage), 58
rollback method (Session), 48, 51
rolling back transactions, 51
  defined, 50
 (NO_ACKNOWLEDGE constant, 41, 49
getAcknowledgeMode method, 48, 49, 50, 69
getMessageListener method, 41, 48
getTransacted method, 48, 50
message creation methods, 56
methods, 47
AUTO_ACKNOWLEDGE constant, 41, 49
CLIENT_ACKNOWLEDGE constant, 41, 49
commit method, 48, 51
Session object
  close method, 23, 25
  close method, 23, 25
send method (MessageProducer), 23, 62, 63
send method (MessageProducer), 23, 62, 63
Session object
  close method, 23, 25
createBrowser method, 48, 70
createBytesMessage method, 48, 56, 60
createConsumer method, 47, 64, 65, 66, 67, 68
Session object
  createConsumer method, 24
Session object
  createDurableSubscriber method, 47, 66, 67, 68
createMapMessage method, 47, 56
createMessage method, 47, 56
createObjectMessage method, 47, 56, 60
createProducer method, 47, 61, 62
Session object
  createProducer method, 22
Session object
  createQueue method, 48
createStreamMessage method, 47, 56, 57
createTemporaryQueue method, 47, 48
createTemporaryTopic method, 47, 48
createTextMessage method, 47, 56, 57
Session object
  createTextMessage method, 22
Session object
  createTopic method, 48
DUPS_OK_ACKNOWLEDGE constant, 41, 49
getAcknowledgeMode method, 48, 49, 50, 69
getmessageListener method, 41, 48
getTransacted method, 48, 50
message creation methods, 56
methods, 47
AUTO_ACKNOWLEDGE constant, 41, 49
CLIENT_ACKNOWLEDGE constant, 41, 49
recover method, 48
rollback method, 48, 51
SESSION_TRANSACTED constant, 50, 69
setMessageListener method, 41, 48
unsubscribe method, 47, 67, 71
SESSION_TRANSACTED constant (Session), 50, 69
sessions
  AUTO_ACKNOWLEDGE constant, 41, 49
  CLIENT_ACKNOWLEDGE constant, 41, 49
  close method, 23, 25
  close method, 48
  closing, 23, 25
  commit method, 48, 51
createBrowser method, 48, 70
createBytesMessage method, 48, 56, 60
createConsumer method, 47, 64, 65, 66, 67, 68
createDurableSubscriber method, 47, 66, 67, 68
createMapMessage method, 47, 56

S
SAAJ, See SOAP with Attachments API for Java
saaj-api.jar file, 18
secure connections, 42
selectors, message
  browsing with, 70
  defined, 65
  efficient use of, 83
  performance impact, 97-98
send method (MessageProducer), 23, 62, 63
send messages, 23, 61
sequencing partial messages, 86
ServletConfig object, 180
Session object, 22, 24, 34
  AUTO_ACKNOWLEDGE constant, 41, 49
  CLIENT_ACKNOWLEDGE constant, 41, 49
  close method, 48
Session object
  close method, 23, 25
Session object
  commit method, 48, 51
createBrowser method, 48, 70
createBytesMessage method, 48, 56, 60
createConsumer method, 47, 64, 65, 66, 67, 68
Session object
  createConsumer method, 24
sessions (Continued)
createMessage method, 47, 56
createObjectMessage method, 47, 56, 60
createProducer method, 22
createProducer method, 47, 61, 62
createStreamQueue method, 48
createTemporaryQueue method, 47, 56, 57
createTemporaryQueue method, 47, 48
createTemporaryTopic method, 47, 48
createQueue method, 48
createStreamMessage method, 47, 56, 57
createTextMessage method, 22
createTextMessage method, 47, 56, 57
createTopic method, 48
creating, 22, 24, 40, 48
defined, 47
DUPS_OK_ACKNOWLEDGE constant, 41, 49
getAcknowledgeMode method, 48, 49, 50, 69
getMessageListener method, 41, 48
getTransacted method, 48, 50
NO_ACKNOWLEDGE constant, 41, 49
recover method, 48
rollback method, 48, 51
SESSION_TRANSACTED constant, 50, 69
setMessageListener method, 41, 48
threading restrictions in, 85
transacted, 22, 24, 50, 69
unsubscribe method, 47, 67, 71
work done by, 81
setBoolean method (MapMessage), 59
setBooleanProperty method (Message), 55, 87
setByte method (MapMessage), 59
setByteProperty method (Message), 55
setBytes method (MapMessage), 59
setChar method (MapMessage), 59
setClientID method (Connection), 40, 41
setDeliveryMode method (MessageProducer), 62, 63
setDisabledMessageID method
(MessageProducer), 53, 62, 63
setDisabledMessageTimestamp method
(MessageProducer), 54, 62, 63
setDoubleProperty method (Message), 55
setEventListener method (Connection), 40
setExceptionListener method (Connection), 40, 41
setFloat method (MapMessage), 59
setFloatProperty method (Message), 55
setInt method (MapMessage), 59
setIntProperty method (Message), 55
setJMSCorrelationID method (Message), 53
setJMSCorrelationIDAsBytes method (Message), 53
setJMSDeliveryMode method (Message), 53
setJMSDestination method (Message), 52
setJMSExpiration method (Message), 53
setJMSMessageID method (Message), 52, 54, 55
setJMSRedelivered method (Message), 53
setJMSReplyTo method (Message), 53
setJMSTimestamp method (Message), 53
setJMSType method (Message), 53
setLong method (MapMessage), 59
setLongProperty method (Message), 55
setMessageListener method (MessageConsumer), 65, 68
setMessageListener method (Session), 41, 48
setObject method (MapMessage), 59
setObject method (ObjectMessage), 60
setObjectProperty method (Message), 54, 55
setPriority method (MessageProducer), 62, 63
setProperty method (AdministeredObject), 37, 39, 46
setProperty method (AdministeredObject), 21, 23
setShort method (MapMessage), 59
setShortProperty method (Message), 55
setString method (MapMessage), 59
setStringProperty method (Message), 55
setStringProperty method (Message), 23
setText method (TextMessage), 57
setText method (TextMessage), 22
setText method (TextMessage), 62, 63
Simple Object Access Protocol (SOAP)
client code, 177
collections, 174
defined, 161
defaults, 173
exception handling, 176
 fault codes, 183
fault handling, 176, 182
layers, 162
message factories, 174
namespaces, 171
Simple Object Access Protocol (SOAP) (Continued)

- point-to-point connections, 175
- programming models, 174
- service code, 179

SOAPMessageFromJMSMessage method, 186
SOAPMessageIntoJMSMessage utility, 186

SOAP, See Simple Object Access Protocol, 161

SOAP messages
- disassembling, 181
- envelope, 165
- header, 165
- MIME envelope for, 166
- models of, 165
- Name object, 173
- payload to JMS message as, 185
- SOAPMessage object, 168
- structure of, 165

SOAP with Attachments API for Java (SAAJ)
- about, 167
- client code, 177
- exception handling, 176
- fault handling, 176, 182
- javax.xml.messaging package, 167
- javax.xml.soap package, 167
- programming model, 162, 167, 174
- service code, 179

SOAP with Attachments API for Java (SAAJ)
- Specification, 7

SQL, 65

SSL service, 42

standard message properties, 55
- JMSAppID, 55
- JMSConsumerTXID, 55
- JMSDeliveryCount, 55
- JMSGroupID, 55
- JMSGroupSeq, 55
- JMSProducerTXID, 55
- JMSRcvTimestamp, 55
- JMSState, 55
- JMSUserID, 55

start method (Connection), 24, 40, 41, 42, 68

start method (Connection), 67

starting
- connections, 24, 41, 67

starting (Continued)
- message brokers, 19

stop method (Connection), 40, 41

stream messages
- composing, 57
- defined, 56
- processing, 74
- readBoolean method, 74
- readByte method, 74
- readBytes method, 74
- readChar method, 74
- readDouble method, 74
- readFloat method, 74
- readInt method, 74
- readLong method, 74
- readObject method, 74
- readShort method, 74
- readString method, 74
- reset method, 58
- writeBoolean method, 58
- writeByte method, 58
- writeBytes method, 58
- writeChar method, 58
- writeDouble method, 58
- writeFloat method, 58
- writeInt method, 58
- writeLong method, 58
- writeObject method, 58
- writeShort method, 58
- writeString method, 58

StreamMessage object, 56

- access methods, 74
- composition methods, 57
- readBoolean method, 74
- readByte method, 74
- readBytes method, 74
- readChar method, 74
- readDouble method, 74
- readFloat method, 74
- readInt method, 74
- readLong method, 74
- readObject method, 74
- readShort method, 74
- readString method, 74

Index
StreamMessage object (Continued)
    reset method, 58
    writeBoolean method, 58
    writeByte method, 58
    writeBytes method, 58
    writeChar method, 58
    writeDouble method, 58
    writeFloat method, 58
    writeInt method, 58
    writeLong method, 58
    writeObject method, 58
    writeShort method, 58
    writeString method, 58
subscriber name, 66
synchronous message consumption, 67
    defined, 64

T
temporary destinations, defined, 47
TemporaryQueue object, delete method, 47
TemporaryTopic object, delete method, 47
testing message brokers, 20-21
text messages
    composing, 57
    defined, 56
    getText method, 25
    getText method, 73, 74
    processing, 73
    setText method, 57
    setText method, 22
TextMessage object, 56
    access method, 73
    getText method, 73, 74
TextMessage object
    getText method, 25
TextMessage object
    setText method, 57
TextMessage object
    setText method, 22
threads, See client threads
time stamp (message)
    defined, 54
    suppressing, 63
time-to-live, See lifetime (message)
topic destinations, 22, 24, 43, 62, 63, 64
    defined, 34
    durable subscribers, 66
Topic object, 34, 43
    constructor method, 46
topic publishers, defined, 62
topic sessions
    createPublisher method, 63
    createSubscriber method, 64
topic subscribers
    defined, 64
    durable, 64, 66, 71
    nondurable, 64, 71
TopicConnection object, 34
TopicConnectionFactory object, 34
TopicPublisher object, 34, 63
TopicSession object, 34, 63, 64
    createPublisher method, 63
    createSubscriber method, 64
TopicSubscriber object, 34, 64, 66
transacted sessions, 22, 24, 50
    and acknowledgment, 69
    defined, 50
transactions
    and custom client acknowledgment, 136
    committing, 50, 51
    defined, 50
    performance impact of, 95
    rolling back, 50, 51

U
unified messaging domain, 40, 43
    defined, 34
unsubscribe method (Session), 47, 67, 71
URL Endpoint object, 179
user authentication, 40

W
warning messages, 203
Web services, 161
writeBoolean method (BytesMessage), 61
writeBoolean method (StreamMessage), 58
writeByte method (BytesMessage), 61
writeByte method (StreamMessage), 58
writeBytes method (BytesMessage), 61
writeBytes method (StreamMessage), 58
writeChar method (BytesMessage), 61
writeChar method (StreamMessage), 58
writeDouble method (BytesMessage), 61
writeDouble method (StreamMessage), 58
writeFloat method (BytesMessage), 61
writeFloat method (StreamMessage), 58
writeInt method (BytesMessage), 60
writeInt method (MapMessage), 59
writeInt method (StreamMessage), 58
writeLong method (BytesMessage), 61
writeLong method (StreamMessage), 58
writeObject method (BytesMessage), 61
writeObject method (StreamMessage), 58
writeShort method (BytesMessage), 61
writeShort method (StreamMessage), 58
writeString method (StreamMessage), 58
writeUTF method (BytesMessage), 61

X
XML schema validation, 138
XSD, See XML schema validation